AFIT/GCE/ENG/91D-06

FORMAL VERIFICATION OF DIGITAL LOGIC

THESIS

Stuart Lewis Labovitz
Captain, USAF

AFIT/GCE/ENG/91D-06

December 1991 Master's Thesis

FORMAL VERIFICATION OF DIGITAL LOGIC

Stuart L. Labovitz, Captain, USAF

Air Force Institute of Technology, WPAFB OH 45433-6583

AFIT/GCE/ENG/91D-06

WL/ELE
Wright-Patterson AFB OH 45433-6543

The most widely used technique for checking the correctness of digital circuit designs is simulation. As the complexity of digital circuits has continued to grow, however, circuit designers have become unable to perform complete simulations of their integrated circuits. Formal hardware verification provides an alternative approach, performing a series of mathematical proofs in order to show that the construction of the circuit from its submodules will result in the intended overall circuit behavior. Papers by Barrow in 1983 and 1984 discuss a PROLOG-based hierarchical formal circuit verification system named VERIFY. **AFIT_VERIFY**, a simple, experimental reverse-engineered version of Barrow's VERIFY system, was produced by Captain Kevin Sparks in 1991. Since that time, a new user interface has been added to the **AFIT_VERIFY** system, as well as the capability to maintain a central repository of standard, previously verified parts. This thesis provides a detailed description of these and other improvements that have been made to Sparks's **AFIT_VERIFY** system.

Formal Verification, Prolog, Hardware Verification

209

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL

AFIT/GCE/ENG/91D-06

FORMAL VERIFICATION OF DIGITAL LOGIC

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

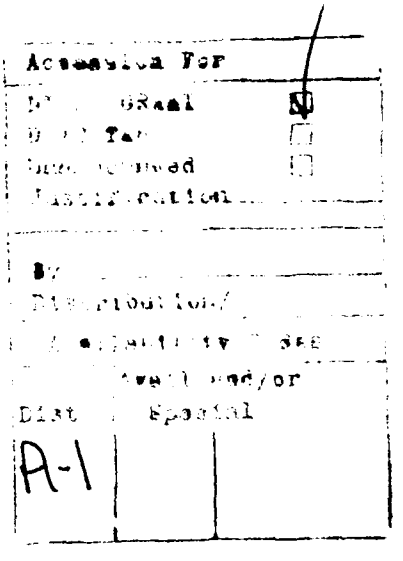
Requirements for the Degree of

Master of Science in Computer Engineering

Stuart Lewis Labovitz, B.S.E.E.

Captain, USAF

December, 1991



Acknowledgments

I would like to thank my wife, Judith, for her understanding, patience, and advice during my time at AFIT. Without her unfailing support and encouragement, this effort would not have been possible. I would also like to thank Doctor Frank M. Brown, my thesis and academic advisor, whose enthusiasm for artificial intelligence, logic programming, and hardware verification was both contagious and enlightening. His support and guidance allowed this project to reach a fruitful conclusion. Doctor Henry Potoczny also provided mathematical guidance whenever required, as well as providing a sounding board for my thoughts, for which I am indebted. I would also like to thank Capt Mark Mehalic for his assistance and guidance in selecting components to be entered into the parts library, and CPT Michael Dukes for his never-ending patience, advice, comments, and assistance on all matters concerning PROLOG and hardware verification.

Stuart Lewis Labovitz

Table of Contents

List of Figures

List of Tables

Abstract

The most widely used technique for checking the correctness of digital circuit designs is simulation. As the complexity of digital circuits has continued to grow, however, circuit designers have become unable to perform complete simulations of their integrated circuits. Formal hardware verification provides an alternative approach, performing a series of mathematical proofs in order to show that the construction of the circuit from its submodules will result in the intended overall circuit behavior. Papers by Barrow in 1983 and 1984 discuss a PROLOG-based hierarchical formal circuit verification system named VERIFY. **AFIT_VERIFY**, a simple, experimental reverse-engineered version of Barrow's VERIFY system, was produced by Captain Kevin Sparks in 1991. Since that time, a new user interface has been added to the **AFIT_VERIFY** system, as well as the capability to maintain a central repository of standard, previously verified parts. This thesis provides a detailed description of these and other improvements that have been made to Sparks's **AFIT_VERIFY** system.

FORMAL VERIFICATION OF DIGITAL LOGIC

I. Introduction

1.1 Background

In digital design, as in any design activity, the goal is to produce a final product which meets its specifications. Making sure that this goal is met can be difficult. As the number of components used in a digital integrated circuit (IC) continues to grow into the millions, the ability to verify that circuit designs are correct becomes even more essential and even more difficult [24:435]. There are a number of possible approaches to solving this problem, including formal synthesis, exhaustive simulation, and formal verification.

Formal synthesis, or the automatic transformation of design specifications into a fully realized design, is not yet practical. Exhaustive simulation, a common alternative that is currently available to the system designer, falls prey to the sheer number of input combinations required to perform this task. As an example, many ICs currently being designed at AFIT have 144 or more input pins. An IC with 144 input pins would require at least 2^{144} or $2.23 * 10^{43}$ different input combinations, and in some cases we would even need to try at least every possible ordering of these input combinations, or (at a minimum) $2^{144}!$ input sequences [11:6]. A simple device that multiplies two 16-bit integers would require the testing of over four billion different inputs, and any circuit that contains a single 32-bit register can have more than four billion different responses to any given input vector [2:64]. Use of non-exhaustive simulation relies upon the skill of the designer and the tester in selecting an appropriate set of input combinations, otherwise known as test vectors, in a proper sequence. Such a simulation, however, will only result in confirmation that the design is partially correct with respect to these particular test vectors [24:435]. Formal verification provides a means of fully verifying systems through the application of mathematical transformations. These transformations are used to produce a mathematical proof that the realized design logically implies the original behavioral specification.

1.2 Problem Description.

This thesis will apply a strategy of hierarchical formal verification to the problem of verification of digital circuits. As stated above, formal verification methods attempt to demonstrate that a behavioral description derived from the high-level design of the physical realization of a circuit design logically implies the high-level behavior contained in the original design specification, as represented in Equation (1.1).

$$\text{Structurally-Derived Behavior} \longrightarrow \text{Specified Behavior.} \tag{1.1}$$

Previous formal verification systems, and specifically the **AFIT_VERIFY** system developed by Captain Kevin Sparks and discussed in Section 3.3.1, have all approached a subset of this problem by proving the equivalence between these two behaviors [30], as represented in Equation (1.2), rather than the logical implication shown above in Equation (1.1).

$$\text{Structurally-Derived Behavior} \longleftrightarrow \text{Specified Behavior.} \tag{1.2}$$

This approach, however, only covers a subclass of circuit designs. If the individual specifying the behavior of a proposed system fully specifies the behavior under all input conditions, the resulting structurally-derived (implementation) behavior will be isomorphic to the specified behavior, as indicated in Equation (1.2). However, the individual specifying the behavior of a proposed system may instead list only a subset of input and output conditions, leaving the remainder as "don't care" conditions. These "don't care" conditions may then be used by the system designer to make the resulting system smaller, faster, or simpler than would be otherwise possible. The resulting implementation behavior, however, is no longer isomorphic to the specified behavior, as indicated in Equation (1.1). The use of "don't care" conditions in system design is common, and therefore a practical formal verification system should ideally allow for verification between derived behaviors and specified behaviors which are not necessarily isomorphic. Isomorphisms, as well as other mathematical concepts, are discussed in more detail in Chapter II.

1.3 Assumptions

1.3.1 Theory. It is assumed that the reader has a basic understanding of programming in PROLOG. An overview of the PROLOG language is provided in Chapter IV, but

this overview is not intended to be complete or comprehensive. If further information on the PROLOG language is required, either *The Art of Prolog: Advanced Programming Techniques* [31] by Leon Sterling and Ehud Shapiro or *Prolog Programming for Artificial Intelligence* [4] by Ivan Bratko are highly recommended. Additionally, the reader should have a basic understanding of the theory of sets and algebras. A basic overview of these topics is provided in Chapter II. If further information on these topics is required, *Introduction to Computer Theory* [6] by Daniel Cohen or *Introduction to Automata Theory, Languages, and Computation* [19] by John Hopcroft and Jeffrey Ullman are recommended.

1.3.2 Resources. This research did not require any special resources. UNIX-based computer systems using Quintus PROLOG which were either currently available in the AFIT School of Engineering or provided by the sponsor of this research, Microelectronics Division, Electronic Technology Directorate, Wright Laboratory (WL/ELE), provided a programming environment adequate to perform the research.

1.4 Scope

The scope of this thesis effort was limited by the amount of time available for program development. This thesis effort included the following modifications and extensions to **AFIT_VERIFY**:

- Addition of the capability to store verified components in a standard cell library
- Enhancement of the user interface, to include a menu-based system for user-to-program interaction
- Continued testing of the capabilities of the **AFIT_VERIFY** system
- Inclusion of new proof strategies into the **AFIT_VERIFY** framework
- Insertion of additional components into the standard cell library.

In addition, an obviously crucial issue to the scope of this work is the meaning of verification. Although this is discussed further later in this document, it is only meaningful to discuss the verification of correctness relative to a specification. The relationship between a specification and its verification should ensure that an implementation exceeds the minimum requirements stated in a specification [9:13-14]. Additionally, since determining the validity of a logic expression is an NP-complete problem [24:435] and answering non-trivial questions about digital circuit properties can be NP-hard (*e.g.*,

requiring solutions that are at least exponential in complexity), it is imperative that a verification methodology use a modular and hierarchical approach. A system is modular when it can be described as a collection of modules with limited and well-defined interfaces. These interfaces should be described such that the module can be connected into a larger system without reference to its internal structure, thereby limiting the complexity of the system. Hierarchical verification means that the specifications of modules at lower levels of abstraction can be used as descriptions of modules at higher levels, thereby allowing the task to be broken into smaller, more easily handled parts [9:13-14]. This thesis is concerned exclusively with verification of modular, hierarchical digital systems.

1.5 Standards

Where practical, programs adhere to the appropriate international standards. This was limited, however, by the lack of an international standard for the PROLOG language. Most implementations of PROLOG, however, adhere to the so-called DEC-10 (otherwise known as Edinburgh, or Clocksin and Mellish) syntax. As a result, the **AFIT_VERIFY** system was written so that it will fully use the facilities provided by Quintus PROLOG, which uses this syntax. Most of the library calls in Quintus PROLOG are supplied with source code, so they could be moved to other DEC-10 (Edinburgh) PROLOG compatible environments.

1.6 Approach/Methodology

This thesis effort has been based upon an ongoing analysis of the state of the **AFIT_VERIFY** system. As this analysis effort progressed, new features were added and errors were slowly eliminated. Since the PROLOG source code for the **AFIT_VERIFY** system is highly interdependent, regression testing was required after all changes in order to determine the ramifications upon the general operation of the system.

Major effort was put into improving the user interface, along with integrating a parts library system into the **AFIT_VERIFY** framework. Extensive effort was also put into coding a number of complex test circuits, many of which helped to identify flaws in the PROLOG source code. In addition, some preliminary work in expanding the proof system was accomplished, but was not integrated into the system due to lack of time. The resulting version of **AFIT_VERIFY** is much more robust, well-documented, and extensively tested than the version produced by Captain Kevin Sparks at the conclusion of his thesis effort.

1.7 Expected Benefits of This Research

The work described in this thesis is an expansion of past efforts to produce a hardware verification tool for use by AFIT students and faculty. Once completed, **AFIT_VERIFY** will allow VLSI designers to perform hardware verification upon portions of their designs, thereby supplementing the SPICE and VHDL simulations of these designs. As the **AFIT_VERIFY** system does not currently deal with temporal dependencies, the need for simulations will not be entirely eliminated, but the reliance upon them will be greatly diminished.

1.8 Chapter Summary and Thesis Overview

Chapter II contains a brief explanation of some of the mathematical theory and terms used elsewhere in the thesis. Chapter III consists of a literature review of relevant work. Some significant systems developed by other researchers are discussed, as well as the previous work performed at AFIT. Chapter IV provides an introduction to the PROLOG language used in developing the **AFIT_VERIFY** system. Chapter V discusses the evolution of **AFIT_VERIFY**, and Chapter VI summarizes the results of this research, along with recommendations for further work. Program source code and sample program runs are provided in the appendices.

II. Mathematical Theory And Concepts

2.1 Introduction

2.1.1 Problem Description. As previously discussed in Chapter I, this thesis applies a strategy of hierarchical formal verification to the problem of verification of digital circuits. This strategy attempts to demonstrate that a system-level behavioral description derived from the modular realization of a circuit design logically implies the high-level behavior contained in the original overall design specification for that system, as represented in Equation (2.1).

$$\text{Structurally-Derived Behavior} \longrightarrow \text{Specified Behavior.} \tag{2.1}$$

Many formal verification systems have approached a subset of this problem by proving the equivalence between these two behaviors, as represented in Equation (2.2), rather than the logical implication shown above in Equation (2.1).

$$\text{Structurally-Derived Behavior} \longleftrightarrow \text{Specified Behavior.} \tag{2.2}$$

The approach described by Equation (2.2), however, only covers a subclass of the problem presented in Equation (2.1). As shown below in Section 2.2.4, Equation (2.2) describes an isomorphic relationship between the system's specified behavior and its structurally derived behavior. Equation (2.1), however, describes a more general homomorphic relationship (discussed further in Section 2.2.3) between the system's specified behavior and its structurally derived behavior.

If the individual specifying the overall behavior of a proposed system fully specifies its behavior under all input conditions, the resulting structurally-derived (implementation) behavior should be isomorphic to the specified behavior. However, if the individual specifying the behavior of a proposed system instead lists only a subset of input and output conditions, leaving the remainder as "don't care" conditions, then these "don't care" conditions may then be used by the system designer to make the resulting system smaller, faster, or simpler than would be otherwise possible. The resulting implementation behavior, however, is no longer isomorphic to the specified behavior. The use of "don't care" conditions in the

design of large integrated circuits is common, and therefore a practical formal verification system should allow for verification between structurally derived behaviors and specified behaviors which are not necessarily isomorphic.

2.2 Sets and Mappings

2.2.1 Many-Sorted Algebras.

Definition 1 *A* **many-sorted algebra** *is a seven-tuple*

$$(S, \alpha, \Sigma, \beta, \Phi, \gamma, G) \tag{2.3}$$

where S is the set of sorts; Σ is the class of all signature sets A_s; the set Φ is the class of all carrier sets; β is a mapping from S to Φ, $\beta: S \mapsto \Phi$; α is the mapping $\alpha: S^* \times S \mapsto \Sigma$; G is the set consisting of all operators; and γ maps the symbols contained in the union of the signature sets into G:

$$\gamma: \bigcup_{S^* \times S} \left[\sum_{s_1 s_2 \ldots s_n, s} \right] \mapsto G \tag{2.4}$$

such that for each $\sigma \in \sum_{\epsilon, s}$, $\gamma(\sigma) \in A_s$, and for

$$\sigma \in \sum_{s_1 s_2 \ldots s_n, s} \tag{2.5}$$

$$\gamma(\sigma): A_{s_1} \times A_{s_2} \times \ldots \times A_{s_n} \mapsto A_s \; [10: 398]. \tag{2.6}$$

In the case of finite-state automata, discussed below in Section 2.3, three distinct sorts are employed: states, inputs, and outputs. Thus, in this case,

$$S = \{\text{states, inputs, outputs}\}. \tag{2.7}$$

A signature set, denoted by

$$\sum_{s_1 s_2 \ldots s_n, s} \tag{2.8}$$

contains the names of the operators with n inputs of types $s_1, s_2, \ldots, s_n$ and output of type s. In a finite-state automaton, some of the non-empty signature sets include [10:392–393]:

$$\sum_{\text{state input state, output}} = \{\text{transition function}\} \tag{2.9}$$

$$\sum_{\text{state, output}} = \{\text{final}\} \tag{2.10}$$

$$\sum_{\varepsilon,\text{state}} = \{\text{initial}\} \tag{2.11}$$

It should be noted that a Boolean algebra is a many-sorted algebra. In this case, there is one carrier set $A_{elements}$, five sorts

$$\sum_{\text{element element, element}} = \{Supremum\} \tag{2.12}$$

$$\sum_{\text{element element, element}} = \{Infinum\} \tag{2.13}$$

$$\sum_{\text{element, element}} = \{Complement\} \tag{2.14}$$

$$\sum_{\varepsilon,\text{ element}} = \{One, Zero\} \tag{2.15}$$

and set of operators G is {Complement, Supremum, Infinum} [10:394].

2.2.2 Mappings. The idea of a *mapping* (or *function*) is common to many branches of mathematics. A mapping is defined in Definition 2.

Definition 2 *A* **mapping** θ *from a set* X *to a set* Y *is a rule or procedure that assigns each element* x *of* X *to a unique element* y *of* Y. *The element* y *of* Y *assigned to a given* x *of* X *is called the* **image** *of* x *under* θ *and is denoted by* $\theta(x)$ *[18:127].*

Definition 3 *Let* θ *be a mapping from* X *to* Y. *Then* X *is the* **domain** *and* Y *is the* **codomain** *of* θ. *The* **image set** *of* θ *is the subset of* Y *consisting of the images* y *of all the elements* x *of* X *[18:127].*

When discussing a mapping θ from X to Y, this mapping can be represented by a number of forms. The expression

$$\theta : X \mapsto Y \tag{2.16}$$

is commonly used to express the concept that "θ maps X to Y."

Definition 4 *Let θ be a mapping such that distinct elements x_1 and x_2 of the domain always have distinct images $\theta(x_1)$ and $\theta(x_2)$; that is, let $x_1 \neq x_2$ imply $\theta(x_1) \neq \theta(x_2)$. Then θ is said to be an* **injection**, *or an* **injective mapping** *[18:130].*

An injection is also called a *one-to-one mapping* from X to Y [18:131].

Definition 5 *If the image set of a mapping θ from X to Y is the same set as the codomain Y, the mapping is said to be* **surjective**, *or a* **surjection** *[18:131].*

Definition 6 *A mapping that is both injective and surjective is said to be* **bijective**, *or a* **bijection** *[18:131].*

A surjection may be called a mapping from X *onto* Y, or an *onto mapping.* A bijection from X to Y is also referred to as a *one-to-one correspondence* between X and Y, or a *one-to-one mapping from X onto Y* [18:132].

2.2.3 Homomorphisms. A homomorphism is defined in Definition 7, where a many-sorted algebra is in Section 2.2.1.

Definition 7 *A mapping θ from a many-sorted algebra G to another many-sorted algebra G' with*

$$\theta(ab) = \theta(a)\theta(b) \text{ for all } a \text{ and } b \text{ in } G \tag{2.17}$$

is a **homomorphism** *from G to G' [18:135][10:414].*

It is important to note that the property of homomorphisms given in Equation (2.17) preserves the structure and operations between the elements of X when they are mapped into Y [10:414]. Thus, if σ is a unary operation that is defined in both X and Y with $\sigma(x_1) = x_2$, then $\sigma(\theta(x_1)) = \theta(x_2)$, and if ρ is a binary operation in both X and Y with $\rho(x_1, x_2) = x_3$, then $\rho(\theta(x_1), \theta(x_2)) = \rho(x_3)$ [18:135].

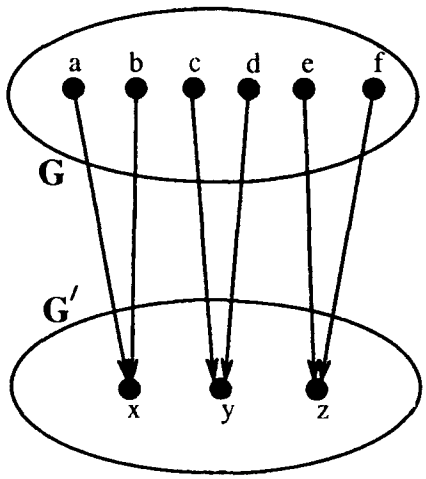


Figure 2.1. Surjective Mapping From G Onto G' [18:129]

Thus, a homomorphism provides a relationship between two algebras, preserving some set of operations on the algebras, but without the restriction that this relationship be an injection. A homomorphism from X to Y is therefore not necessarily an invertible operation, as some of the information contained in X may be "lost" in Y. The surjective mapping from G onto G' shown in Figure 2.1 shows how set G can be mapped onto set G, but there is no function that maps G' onto G. This will also be seen in the homomorphism shown in Figure 2.4 in Section 2.3.

2.2.4 Isomorphisms. An isomorphism is a type of homomorphism, and is formally defined as follows:

Definition 8 *A bijective homomorphism is called an* **isomorphism**; *that is, an isomorphism from G to G' is a bijection θ such that $\theta(ab) = \theta(a)\theta(b)$ for all a and b in G [18:135].*

An isomorphism therefore possesses all of the properties of a homomorphism, as discussed in Section 2.2.3, but with certain added restrictions. Since an isomorphism is a bijection, it provides a one-to-one correspondence between the two algebras. Thus, an isomorphism provides a relabeling between the states and transitions (operations) in one algebra and another. Figure 2.2 shows an example of an isomorphism between the real numbers $\Re$ and the positive real numbers $\Re^+$ given by $\theta(r) = 2^r$ [18:130]. A reverse mapping λ can be found to map $\Re^+$ back to $\Re$, namely $\lambda(r') = (\log r')/(\log 2)$.

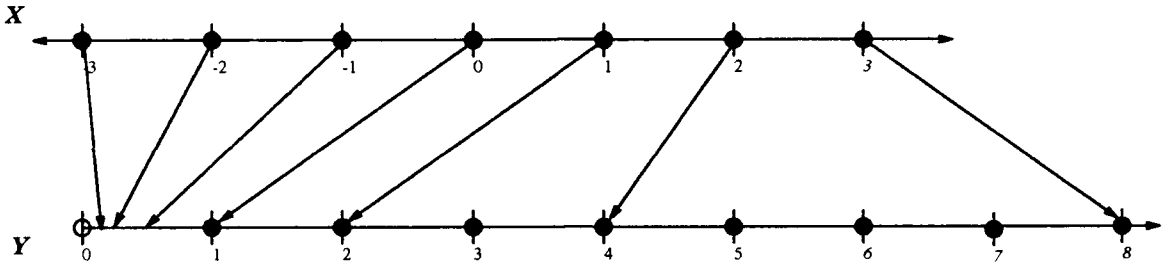


Figure 2.2. Isomorphism From $\Re$ To $\Re^+$ [18:130]

2.3 Finite-State Automata

A finite-state automaton (FSA) is a particular variety of finite-state machine, as shown in Definition 9 [21:318].

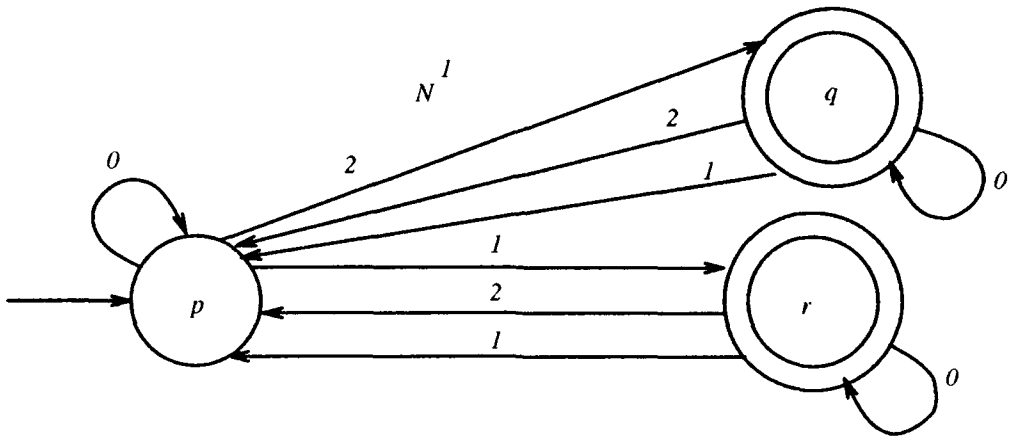
Definition 9 *A* **finite-state automaton** $A = (\Sigma, \mathcal{S}, \delta, \mathcal{A}, \sigma^*)$ *is a finite state machine with a finite set* Σ *of input symbols, a finite set* $\mathcal{S}$ *of states, a next-state function* $\delta : \Sigma \times \mathcal{S} \mapsto \mathcal{S}$*, a subset* $\mathcal{A} \subseteq \mathcal{S}$ *of accepting states, and an initial state* $\sigma^* \in \mathcal{S}$ *[21:316-317].*

Stated otherwise, a FA is a collection of three things:

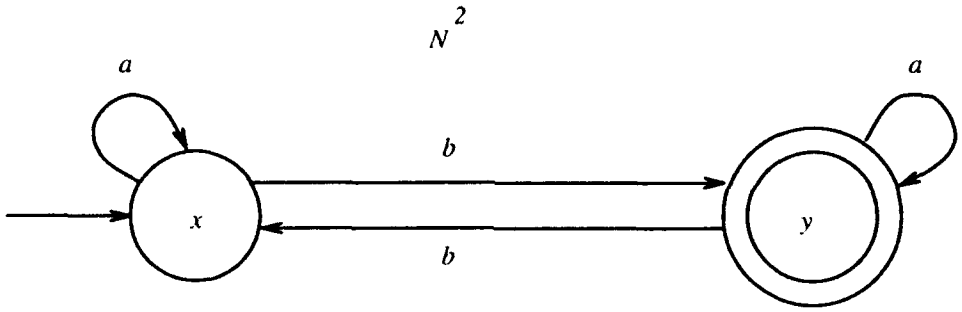
1. A finite set of states $\mathcal{S}$, one of which is designated as the initial state σ^*, called the start state, and some (maybe none) of which are designated as final (accepting) states in set $\mathcal{A}$.
2. An alphabet Σ of possible input letters, from which are formed strings, that are to be read one letter at a time.
3. A finite set of transitions δ that tell, for each state and a given letter of the input alphabet, which state to go to next [6:65].

When a string is input to a FSA, it will end at either an accepting state or a nonaccepting state. If the string ends at an accepting state, we say that the string is *accepted* by the FSA [21:318].

A nondeterministic finite-state automaton (NDFSA), defined in Definition 10, is similar to the FSA discussed above. In the NDFSA, however, the next-state function δ does not necessarily take us to a uniquely defined state, as would the FSA [21:335]. A string is accepted by a NDFSA if the string can follow any path from the initial state



(a) First Finite-State Automaton



(b) Second Finite-State Automaton

Figure 2.3. Homomorphic Finite-State Automata [10:419-421]

to an accepting state [21:336]. However, it can be shown (though application of Kleene's Theorem) that any NDFSA can be represented by an equivalent FSA, and thus, both are of equal power [6:145]. Thus, although there is an important distinction between NDFSA and FSA, this distinction can generally be ignored in all further discussion in this document.

Definition 10 *A* **nondeterministic finite-state automaton** $A = (\Sigma, \mathcal{S}, \delta, \mathcal{A}, \sigma^*)$ *is a finite state machine with a finite set* Σ *of input symbols, a finite set* $\mathcal{S}$ *of states, a next-state function* $\delta : \Sigma \times \mathcal{S} \mapsto \mathcal{P}(\mathcal{S})$, *a subset* $\mathcal{A} \subseteq \mathcal{S}$ *of accepting states, and an initial state* $\sigma^* \in \mathcal{S}$ *[21:334-335].*

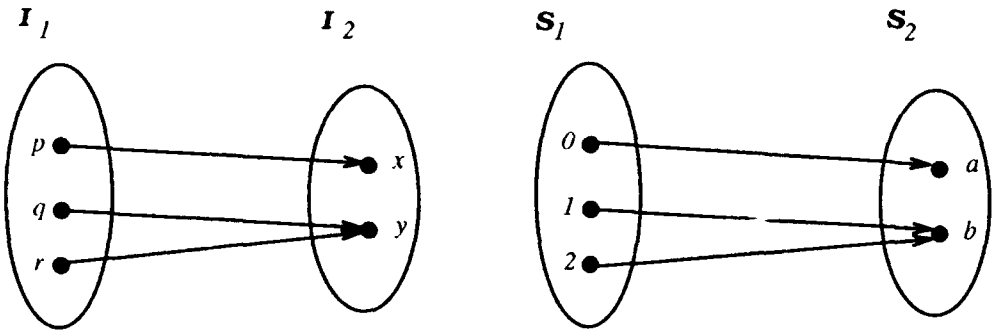


Figure 2.4. Mapping Between Two Homomorphic Automata [10:420]

Since a FSA is a variety of finite-state machine, and thereby symbolizes a set of inter-relationships (transitions) between its elements (states), we can see that the FSA can represent the elements and operations associated with a algebra, as defined in Section 2.2.1. As a result, we can define a homomorphism between one FSA and another. We can show, for example, that there is a homomorphism between the FSA shown in Figure 2.3(a) and that shown in Figure 2.3(b). If we use the mappings shown in Figure 2.4, it can be easily seen that the relationships shown in the FSA in Figure 2.3(a) have been preserved in the FSA in Figure 2.3(b) [10:420–421]. Similarly, isomorphisms can be defined between FSAs. In this case, as in all isomorphisms, the mapping between the FSAs is reduced to an exercise in relabeling the states and transitions.

Finite-state automata are commonly used to model the behavior of simple computational processes. In the mathematical study of finite automata, it has been shown that all deterministic finite-state automata (FSA) (and, for that matter, all non-deterministic finite-state automata (NDFSA)) induce a right invariant equivalence relation R on their set of input strings Σ^*[19:65–71]. This is demonstrated by Theorem 1, the Myhill–Nerode Theorem. (The proof of this theorem is omitted from this document, but can be found in the text by Hopcroft and Ullman [19:65–66].)

Theorem 1 (Myhill–Nerode Theorem) *The following three statements are equivalent:*

1. *The set $L \subseteq \Sigma^*$ is accepted by some finite-state automaton.*
2. *L is the union of some of the equivalence classes of a right invariant equivalence relation of finite index.*

3. *Let equivalence relation R_L be defined by: xR_Ly if and only if for all z in Σ^*, xz is in L exactly when yz is in L. Then R_L is of finite index [19:65–66].*

Theorem 1 can be shown to have, among other consequences, the implication that there is an essentially unique minimum-state FSA for every regular set, as shown in Theorem 2.

Theorem 2 *The minimum state automaton accepting a set L is unique up to an isomorphism (i.e., a renaming of the states)[19:67].*

Thus, if we are able to represent the behaviors of our modules as finite automata, then we should be able to apply Theorems 1 and 2 in order to produce the minimal forms of these automata. The application of isomorphic verification techniques, such as those currently used in the **AFIT_VERIFY** system, could then be used to prove the equivalence between the structural and behavioral specifications. If pure equivalence is not demonstrated, other mathematical techniques could be used to check for the logical implication of the behavioral specification from the structural specification.

III. Literature Review

3.1 Introduction

As discussed in Chapter I, a goal of the digital circuit design process is to produce a description of an integrated circuit (IC) whose operation meets its specifications. Making sure that this goal is met can be difficult. As the number of components used in digital ICs continues to grow, the ability to verify the correctness of circuit designs against their specifications becomes even more essential and even more difficult.

One must also be careful to understand what is meant when a circuit has been "verified." Ideally, the goal of verification is to prove that a physical circuit correctly implements its intended behavior under all circumstances. In reality, however, we can not even attempt this task, as we can not apply logical reasoning to either chips or intentions.

> "The intended behavior rests in the mind of the architects and is not itself accessible. It can be reported in a formal language, but not with checkable accuracy. At the same time, a material device can only be observed and measured, not verified. It can be described in an abstracted way, and the simplified description verified, but again, there is no way to assure the accuracy of the description [8:9]."

Thus, the verification process involves the comparison and manipulation of two or more models of a circuit, where each of these models is a (possibly imperfect) representation of the either the intended design or the final physical realization [8:9]. The realization that these models are merely (imperfect) representations of the real world is especially important when working with "critical systems" whose improper function may result in loss of life or compromise of national security interests. It is important to understand, however, that the fact that a system has been verified in no way insures that this system is safe or that it functions properly under all conditions. It is inherently impossible to apply the certainty of mathematical truths to the analysis of models of objects in the real world [14:223].

3.2 Early Research

3.2.1 Barrow's VERIFY *System.* Barrow describes VERIFY, a Prolog system for using formal verification to ensure the correctness of digital designs [3:437]. In many ways,

the creation of a VERIFY-like system seems to have become the Holy Grail of formal hardware verification. Many subsequent digital hardware verification systems have been based, to a large degree, upon Barrow's description of VERIFY, even though the actual code for the VERIFY system was not included in the text of his article.

Barrow's VERIFY system is based upon the procedure of deriving a behavior from the structural description of a digital system and then comparing that derived behavior to a specified behavioral description for the system. If the two behaviors can be shown to be equivalent, then the structure has been verified to correspond to the specified behavior [3:439]. The key principle used is that, given the behaviors of the components of a system and their interconnections, it is possible to derive a description of the behavior of the overall system, which may then be compared with the original system specifications [1:17]. VERIFY is implemented in PROLOG [3:440] and makes use of the pattern matching abilities of this computer language.

A hardware design in VERIFY consists of a collection of modules that are connected together hierarchically and ultimately constructed from a small set of primitive circuit elements whose structure implicitly defines their behavior. Each module is described as a finite-state machine possessing input ports, output ports, and internal state variables, each having a specified signal type. The behavior of the module is described by two sets of equations defining the output signals (in terms of the input signals and the current state variables), and the new internal states (in terms of the inputs and the current state variables) [2:65]. When VERIFY examines a module for verification, the module is recursively decomposed into its constituent submodules, each of which is turn subjected to verification, thereby deriving an internal representation of the behavior implied by the entire structural description [3:445–446]. VERIFY then attempts to prove an equivalence between the derived behavior and the specified behavior.

VERIFY is designed to take an "intelligent" approach to the verification process. Before any actual verification is accomplished, a series of basic design checks is run on the module to ensure syntactic correctness [3:449]. Once all of the design checks have been passed, the derived behavioral description is obtained by gathering together all of the component behavioral equations. The combined set of all these equations is manipulated algebraically in order to produce an equation which provides an overall derived behavioral description of the system. The derivation of this equation is greatly simplified by not permitting loops in the design, unless they are broken by a state variable, as specified by the Mead-Conway circuit design methodology [2:66]. Now that VERIFY possesses both

a structurally-derived behavior and a specified behavior, it attempts to prove that the two are equivalent [3:451–467]. The most basic case occurs when the two finite-state machines being compared are identical. This case is not as simple as it may seem, as mathematical knowledge of the relationships in the particular system may be required to perform this proof. However, Barrow found that it was possible to write rules for the most commonly occurring situations. In these cases, VERIFY can prove the equivalence between the two isomorphic equations (exact equivalence). In more complex cases, however, the correspondence between the finite-state machine for the structurally-derived behavior and that for the specified behavior may be a homomorphism. These homomorphisms be either structural (when the two machines are effectively functionally identical, but constructed differently) or behavioral (when the two finite-state machines represent two views of a single state-machine, but viewed with different time-scales). VERIFY was designed to handle certain behavioral homomorphisms. Since structural homomorphisms can require in-depth knowledge of the required transformation, however, VERIFY was not designed to prove equivalence between structural homomorphisms [3:451–452].

3.2.2 A Correct By Construction IC Design System. Grabowiecki *et al.* discuss the development of a PROLOG and Turbo-Pascal based environment for the structured design of Very Large Scale Integration (VLSI) integrated circuit systems. This environment presents the user of an IBM PC/XT system with four major subsystems (a text editor, a VLSI floorplan editor, a VLSI layout editor, and a verification program), as well as access to a number of system libraries for predefined PROLOG functions and VLSI standard cells. The floorplan editor is used to create and refine hierarchical designs for integrated circuits, allowing the designer to refine the structure and overall floorplan of the IC as it evolves. The layout editor provides the designer with a method of specifying the structure and geometry of the physical design elements of the IC [15:37]. The verification program, however, is the main focus of this review.

The verification program used in their environment is written in PROLOG and is strongly based upon the pioneering work of Barrow [3]. It is tightly integrated into the entire development environment, enabling an overall top-down design process. Each module is composed of a number of submodules, each of which may, in turn, be composed of other submodules. Modules which are simple enough to define their own layout are labeled leaf cells; these are the primitive elements for the various subsystems of the development environment. When designing a system, the user defines a structure and floorplan for each module, reiterating this process as the module is hierarchically decomposed into its

constituent submodules [15:37–38]. The structure of a module is thereby determined by the interconnections of its submodules.

Grabowiecki's design process follows an iterative path. Behavioral specifications and structural specifications are hierarchically decomposed, with verification between the two occurring at each level of decomposition. If discrepancies are detected, the user is directed to correct them. If none is found, the user is allowed to continue the decomposition, possibly until the system has been decomposed into a collection of leaf cells. This lowest-level hierarchical description of the system is then fed to the layout editor for placement and further refinement [15:38].

The verification subsystem models every module as a finite-state machine with input-vector $\mathbf{x}$, output-vector $\mathbf{z}$, and state-vector $\mathbf{y}$. The outputs and state variables are defined by the equations

$$\mathbf{z}(n) = f(\mathbf{x}(n), \mathbf{y}(n)) \tag{3.1}$$

$$\mathbf{y}(n+1) = g(\mathbf{x}(n), \mathbf{y}(n)) . \tag{3.2}$$

In this notation, $\mathbf{x}(n)$ represents the value of input-vector $\mathbf{x}$ at discrete time n, and $\mathbf{x}(n+1)$ its value at the successive clock time, $n+1$. Although this representation may not be convenient for the circuit designer, it provides a uniform method for specifying the behavior of all digital systems of interest to the authors [15:39].

The proof of equivalence between the specified and derived behaviors is the heart of the verification subsystem. This equivalence may take the form of either an isomorphism or a homomorphism. This system concentrates upon the identification of isomorphisms rather than performing the more complex identification of homomorphisms between the two input sets [15:40–41]. This verification subsystem, however, is intrinsically limited by the implementation decision to provide verification only for systems which can be transformed into isomorphic pairs. Regardless of this limitation, however, the overall design environment appears to be outstanding. Research at AFIT should ideally aim at producing such an integrated design environment.

3.2.3 The VERDIS *Environment for Design Verification.* Brezocnik *et al.* discuss VERDIS, a PROLOG-based system for formal verification which applies rigorous formal mathematical methods to a hierarchically organized description of a digital system [5:100].

The digital system is internally modeled as a finite-state machine $A = \{X, Z, Y, \delta, \lambda\}$, where X and Z are input and output states, respectively, and Y represents a finite set of internal states. The functions δ and λ are the mappings

$$\delta: X \times Y \mapsto Y \tag{3.3}$$

$$\lambda: X \times Y \mapsto Z \tag{3.4}$$

that define the successor internal state and present output signal, respectively, as functions of the current input signal and the current state. VERDIS provides a library of arithmetic, logical, control (*if, case*), and "special" (*fetch, store, joinfn*) functions which can be used in composing the δ and λ functions. All of these functions, however, are actually written using PROLOG [5:101].

The VERDIS system provides a structured format for the description of a system. Each non-primitive module is given a specification (which describes the intended behavior of the module) and an implementation (which describes the construction of the module from more primitive modules). Gate-level primitive modules such as *inverter, nor_gate, nand_gate* are provided in a system library. VERDIS takes full advantage of the declarative syntax of PROLOG, allowing the specification of N-bit components [5:101–102].

VERDIS, like the other hardware verification systems discussed, is based upon the principle that, given the behavior and interconnections of component modules, it is possible to derive a behavioral description of an entire module. This derived behavior is then compared to the specified behavior. This system will identify errors of design, but obviously does not address errors in specification [5:102]. (The designer is free, as always, to specify the wrong system and then implement it — in fact, this seems to happen quite often in military systems!) VERDIS performs some basic consistency checks upon the system before performing the actual verification [5:103]. However, the verification process itself centers around the strategies used in performing the formal proof of equivalence between the specification and the implementation.

VERDIS implements a number of different strategies for proving behavioral and structural equivalence. As in the reviewed articles by Barrow [3] and Grabowiecki [15], VERDIS currently only deals with isomorphic equivalences between the two equations. The authors state that the majority of verification problems can be manipulated so that they yield isomorphic transformations [5:103]. Additional functionality could be added to

a VERDIS-like verification system by allowing the correspondence between automata to be a structural or temporal homomorphism. Continuing work at AFIT should proceed along these lines, as there are many circuits which can only be verified by means of homomorphisms. The VERDIS system proves that the derived and specified behaviors are equivalent by forming an equation between the specified and derived behaviors, and then demonstrating that this equation is an identity for each output and each internal state. This equation is first examined to see if it forms a trivial identity (or trivial non-identity). If the equation is non-trivial, a verification strategy is then selected by the user [5:103]. An interactive interface, such as that provided by VERDIS, along with the additional feature that the user be able to enter new verification strategies, is a worthwhile goal for my hardware verification system.

3.3 Recent Work at AFIT

3.3.1 The **AFIT_VERIFY** *System.* This system, described in the thesis written by Capt Kevin Sparks, is a prototype reverse-engineered version of Barrow's VERIFY program [3]. **AFIT_VERIFY** was written in PROLOG and attempts to determine if supplied structural and behavioral descriptions of a logic circuit are equivalent. Development was relatively successful, and Sparks was able to successfully verify two examples, an integer counter and a full-adder, that are described in Barrow's article [30:Ch 5, 1].

AFIT_VERIFY was initially implemented on an IBM PC/AT compatible system using PROLOG-1, but was later moved to a MicroVax using Quintus PROLOG due to the inherent limitations of PROLOG-1. The construction of the full-adder from 11 primitive (NAND) gates in two hierarchical levels using 30 inter-cell connections exceeded the memory capacity of PROLOG-1, necessitating the use of Quintus PROLOG. The Quintus PROLOG version provided between a 5X and 15X speedup when compared to the PROLOG-1 version, as well as allowing the verification of the full-adder [30:Ch 5, 2].

While Sparks's **AFIT_VERIFY** is only a small prototype system, it successfully demonstrates the ability to verify small circuits. A number of additional features, such as the ability to save a library of previously verified cells, to input VHDL input files directly into the **AFIT_VERIFY** environment, and to have an effective user interface, need to be added to the program in order to make it more suitable for practical use [30:Ch 5, 3].

3.4 Conclusions

Formal verification appears to be both reasonable and necessary in an approach to the problems inherent in VLSI circuit design. A number of systems have been designed over the past seven years, with much of the major work being based upon the pioneering research of Barrow [3]. Recent work at AFIT by Sparks [30] has attempted to adapt this methodology to the needs of the AFIT VLSI community. Continued research is necessary in order to develop Sparks's **AFIT_VERIFY** system into a fully functional tool for formal verification of complex VLSI circuit designs.

IV. Introduction to PROLOG

PROLOG and Lisp are the two primary languages for Artificial Intelligence programming. As in the Middle Ages, when knowledge of Latin and Greek was essential for all scholars, today's practitioners of Artificial Intelligence must be comfortable with both of these languages [4:vii]. Lisp is a functional programming language, concerned with specifying the "how" of a program. Prolog, on the other hand, is a relational, or declarative, language, concerned with describing the "what" of a program by identifying the relationships between pieces of knowledge. As a result, PROLOG is much more suited to the task of hierarchical verification than Lisp, as this task draws heavily upon PROLOG's goal-driven backward-chaining approach and its pattern-matching relational abilities.

4.1 Development of PROLOG

PROLOG is short for *Programmation en Logique*, or *Programming in Logic* [23:38]. In the early 1970s, Robert Kowalski (University of Edinburgh), Alain Colmerauer (University of Marseille-Aix), and Maarten van Emden (University of Edinburgh) provided the initial work on PROLOG, developing its theoretical foundations and providing an initial experimental demonstration of its features [4:xi]. Colmerauer's interest was in natural language processing, while Kowalski's was in logic and theorem-proving [7:26].

Kowalski was developing the concept of logic programming, where a declarative statement of the form

$$P \text{ if } Q \text{ and } R \text{ and } S \tag{4.1}$$

could be interpreted as the statement

$$\text{To solve } P, \text{ solve } Q \text{ and } R \text{ and } S \tag{4.2}$$

and then solved using procedural techniques [31:xi]. Since Equation (4.1) is a Horn clause (discussed further in Section 4.2), Kowalski showed that any first-order predicate-logic statement that can be represented as a set of Horn clauses can be executed as a set of procedures of a recursive programming language using the form of Equation (4.2).

At the same time that Kowalski was doing his work, Colmerauer and his colleagues Henri Meloni and Gerard Battani developed a specialized theorem prover written in Fortran

on an IBM platform [31:xi]. This theorem prover, named PROLOG, embodied Kowalski's procedural interpretation of logic programming. Van Emden and Kowalski later developed formal semantics for the language of logic programs, showing that the operational and model-theoretic semantics were equivalent [31:xxi].

These previous successes led David H.D. Warren of the University of Edinburgh to develop an implementation of PROLOG running on a DECsystem-10 during the mid 1970s [4:xi]. The PROLOG-10 compiler was highly efficient, dispelling many of the myths that had built up concerning the impracticality of logic programming. This compiler, which was itself almost entirely written in PROLOG, showed list-processing performance comparable to that of the best Lisp systems of the time and showed that classical programming, as well as Artificial Intelligence, can benefit from the use of logic programming techniques. As a result, the Edinburgh PROLOG standard that resulted has become a defacto standard for the PROLOG language [31:xxii] and has led to the present popularity of the language. PROLOG is currently available for MS-DOS, Amiga, Unix, VAX/VMS, Macintosh, and other environments.

4.2 PROLOG *Syntax*

PROLOG represents knowledge as first-order predicate logic written in Horn clause form [4:61]. A Horn clause is a logical implication with no more than a single atom as a consequent, where an atom represents some indivisible concept. A Horn clause can be represented as

$$C \Leftarrow A_1 \wedge A_2 \wedge \ldots \wedge A_n \; . \tag{4.3}$$

Use of the Horn clause form of first-order predicate logic allows PROLOG to perform a proof by refutation of user-supplied theorems through the application of Robinson's Resolution Principle [29].

4.2.1 Basic Structures. PROLOG has three basic statement, or *clause*, types: facts, rules, and queries. All three are constructed from PROLOG's single data type, the logical term [31:2] and must end in a period. Terms come in three forms: the *constant*, the *variable*, and the *structure*. *Constants* are either atoms or numbers, where an atom is any sequence of alphanumeric characters that either begins in a lowercase letter or is enclosed in single quotes [4:30]. *Variables* are represented by a sequence of alphanumeric characters that begins with either an underscore character or an uppercase letter. *Anonymous variables*

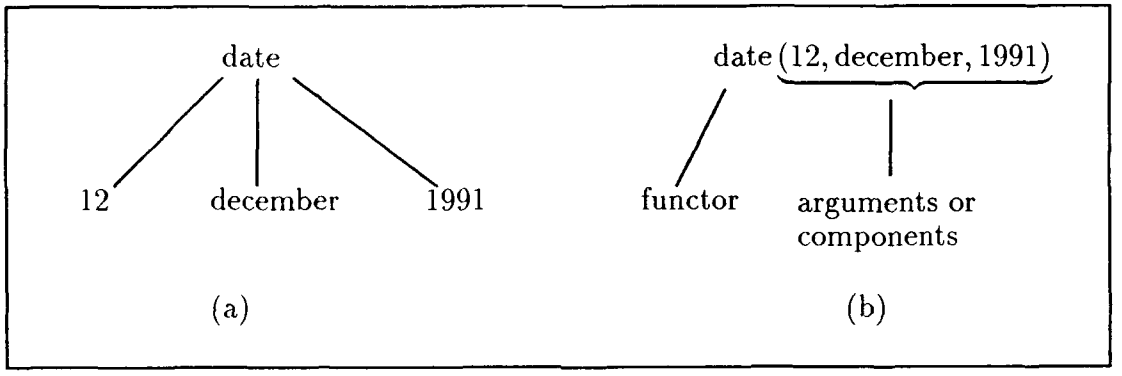


Figure 4.1. Date as an example of a structured object: (a) represented as a tree; (b) as it is written in PROLOG [4:33].

represent "don't care" conditions in the execution of a clause. Edinburgh PROLOG implementations traditionally represent anonymous variables by using the underscore character ('_') as the variable name [31:199]. Thus, an anonymous variable can be used whenever the particular value of the variable does not matter, but the existence of *some* instantiated value is necessary. The lexical scope of all variables is one clause [4:33], and all instances of anonymous variables are considered to be unique.

Structures are objects which contain other components, which may themselves be structures. For example, a date can be viewed as a structure with three components: day, month, and year. In order to treat a structure as a single object, we identify it by an atom, called the *principal functor*, whose arguments are the component terms. In our example, the functor `date` would express a relationship between the day, month, and year components [4:33]. The structure is viewed by PROLOG as a tree, where the root is the functor and each child is a component, which may be a subtree if the component is also a structure [4:34]. An example of this is shown in Figure 4.1. Functors are commonly referred to by their name and arity (the number of arguments). Thus, in our example, we have defined the structure of the functor `date/3`.

The simplest type of PROLOG clause is the *fact*. Facts state relationships among objects [31:2], announcing that the given relationship is true. Some examples of facts are shown in Table 4.1. A finite set of facts can constitute the simplest form of a PROLOG *program*. Table 4.2 provides a sample program that will be used in this section [31:3].

PROLOG fact	Interpretation of the PROLOG fact
father(abraham,isaac).	Abraham is the father of Isaac
mother(sarah,isaac).	Sarah is the mother of Isaac
plus(0,0,0).	$0 + 0 = 0$
plus(1,1,2).	$1 + 1 = 2$
plus(1,3,4).	$1 + 3 = 4$
primitive_component(nand).	A NAND gate is a primitive component
course(csce756,1600_hrs).	CSCE 756 is scheduled at 1600 hours
triangle(point(1,1),point(1,2),point(2,3)).	A triangle containing the points (1,1),(1,2),(1,3)

Table 4.1. Examples of PROLOG facts and their meanings.

father(terach,abraham).	male(terach).
father(terach,nachor).	male(abraham).
father(terach,haran).	male(nachor).
father(abraham,isaac).	male(haran).
father(haran,lot).	male(lot).
father(haran,milcah).	male(isaac).
father(haran,yiscah).	female(sarah).
mother(sarah,isaac).	female(milcah).
	female(yiscah).

Table 4.2. Sample PROLOG Biblical family database [31:3]

A *rule* specifies things that are true if some other condition in the database is satisfied [4:10]. Thus, rules allow us to define new relationships in term of previously existing relationships. Rules are given by statements of the form

$$A \leftarrow B_1, B_2, \ldots, B_n \tag{4.4}$$

where $n \geq 0$. A is known as the *head* of the rule, and the B_i's are known as the *goals*, or the *body*, of the rule. In the case when $n = 0$, the rule degenerates into a fact [31:8-9]. Adjacent goals may be separated by a comma (','), representing a logical AND, or a semicolon (';'), representing a logical OR. PROLOG uses the symbol :- to represent the

$\leftarrow$ shown in Equation (4.4). Thus, a rule [31:9] such as

$$\text{grandfather(X,Y)} \leftarrow \text{father(X,Z), father(Z,Y)} \tag{4.5}$$

would be written instead [4:11] as

$$\underbrace{\texttt{grandfather(X,Y)}}_{\text{head}} \texttt{ :- } \underbrace{\texttt{father(X,Z), father(Z,Y).}}_{\text{body}} \tag{4.6}$$

when entered as part of a PROLOG program. Thus, a PROLOG program consists of a finite set of facts and rules. Since the set of facts and rules in the program can change as the program executes, the current set of facts and rules at any time is often referred to as *working memory* or the PROLOG *database*.

Since the head of a rule is a structure, rules (and thereby facts) are also classified by their name and arity. In Equation (4.6), we have defined the functor `grandfather/2`, and in Table 4.1, we have defined the functors `father/2`, `mother/2`, `plus/3`, `course/2` , `primitive_component/1`, `point/2`, and `triangle/3`. When a rule is written as multiple clauses with the same principal functor and arity, as is `predecessor/2` in Table 4.3, then the PROLOG interpreter performs a logical AND of the clauses, resulting in the same outcome as if they were written in a single clause with the various clause bodies separated by semicolons.

A *query* is a means of retrieving information from a program by asking whether a given relationship exists among a collection of objects. Queries are answered by comparing the query to the information (rules and facts) currently stored in the database, and can thus be viewed as proposed facts whose validity is to be tested against the database. Queries are usually supplied to the prompt of the runtime PROLOG interpreter, and are typically written in the form given by

```
?- mother(sarah,isaac).
```
(4.7)

Thus, for the information shown in Table 4.2, the queries *male(lot)?* and *father(haran,lot)?* can be matched against information in the database and will return the answer *yes*. The queries *male(sarah)?* and *mother(abraham,isaac)?* will return the answer *no*, as this

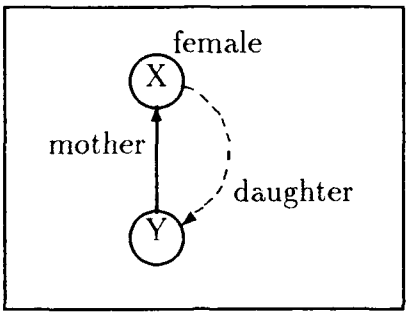


Figure 4.2. Definition graph for the relation **daughter** in terms of other relations

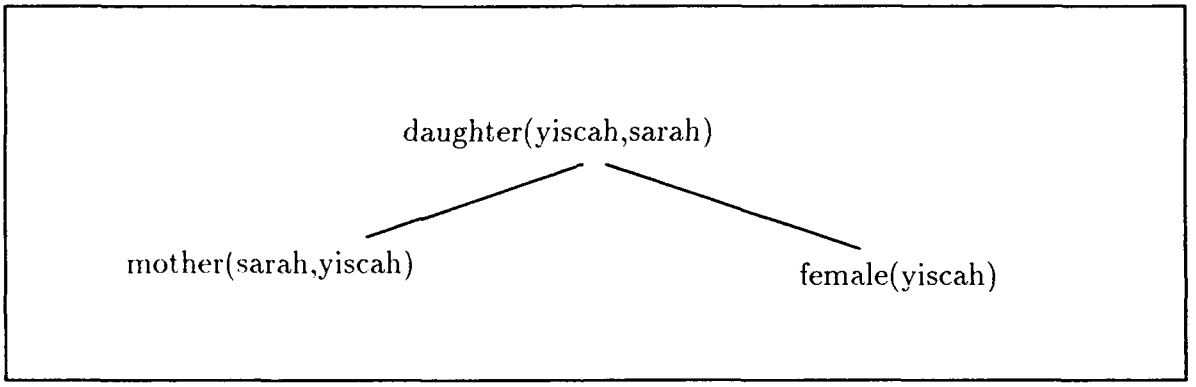


Figure 4.3. A simple proof tree for the query *?- daughter(yiscah,sarah)*

information can not be found as a fact in the database or deduced from any rules in the database [31:3-4].

4.2.2 Simple Examples of PROLOG *Code.* After adding the rule

```
daughter(X,Y) :- mother(Y,X), female(X).        (4.8)
```

to the database shown in Table 4.2 [31:13], one can then ask the query

```
?- daughter(yiscah,sarah).                      (4.9)
```

```
sister(X,Y) :-          /* For any X and Y, X is a sister of Y  */
   female(X),           /*  (1) X is female                     */
   parent(Z,X),         /*  (2) both X and Y have the same      */
   parent(Z,Y),         /*      parent, Z                       */
   X \== Y.             /*  (3) X and Y are different people.   */

mother(X) :-            /* Any person X is a mother if she is   */
   mother(X,_Y).        /*  the mother of some child _Y.        */

predecessor(X,Z) :-     /* Person X is an ancestral predecessor */
   parent(X,Z).         /*  of Z if X is a parent of Z.         */
predecessor(X,Z) :-     /* Person X is an ancestral predecessor */
   parent(X,Y),         /*  of Z if X was the parent of Y, who  */
   predecessor(Y,Z).    /*  is an ancestral predecessor of Z.   */

gcd(X,X,X).             /* Greatest common divisor (gcd) of     */
                        /*  any number X with itself is X.      */
gcd(X,Y,D) :-           /* To find gcd, D, of any X and Y, if   */
   X < Y,               /*  X < Y, compute the new value of     */
   Y1 is Y - X,         /*  Y1=Y-X, and then compute the        */
   gcd(X,Y1,D).         /*  gcd of X and Y1.                    */
gcd(X,Y,D) :-           /* To find gcd. D, of any X and Y, if   */
   X > Y,               /*  X > Y, compute the new value        */
   X1 is X - Y,         /*  of X1=X-Y, and then compute the     */
   gcd(X1,Y,D).         /*  gcd of X1 and Y.                    */
```

Table 4.3. Examples of well constructed PROLOG facts and rules

This causes a search of the current PROLOG database, in order, from the first clause to the last. Since there are no facts referring to the functor daughter/2, PROLOG can not find a solution to the query by direct examination. However, the search of the database locates the rule shown in (4.8), which has two goals, *mother(Y,X)* and *female (X)*, where the variables *Y* and *X* are unified with *sarah* and *yiscah*, respectively. The database is then searched for *mother(sarah,yiscah)*. This existential query is confirmed by a matching fact in the database. PROLOG then attempts to prove *female(yiscah)*, which is also confirmed by a fact in the database. Since all of the goals associated with the body of (4.8) have been successfully proved, PROLOG then concludes that the head is true, returning the answer *yes* to the user [4:11]. The rule shown in (4.8) can be represented by a definition graph [4:12], as shown in Figure 4.2, and the proof tree for the query discussed above [31:14] is shown in Figure 4.3.

```
predecessor_2(X,Z) :-   /* A person X is an ancestral          */
   parent(X,Z).         /*  predecessor of Z if X is a parent  */
                        /*  of Z.                              */

predecessor_2(X,Z) :-   /* A person X is an ancestral          */
   parent(X,Y),         /*  predecessor of Z if X was the      */
   parent(Y,Z).         /*  parent of Y, who is a parent of Z. */

predecessor_2(X,Z) :-   /* A person X is an ancestral          */
   parent(X,A),         /*  predecessor of Z if X was the      */
   parent(A,Y),         /*  parent of A, who is a parent of Y, */
   parent(Y,Z).         /*  who is a parent of Z.              */

predecessor_2(X,Z) :-   /* A person X is an ancestral          */
   parent(X,A),         /*  predecessor of Z if X was the      */
   parent(A,B),         /*  parent of A, who is a parent of B, */
   parent(B,Y),         /*  who is a parent of Y, who is a     */
   parent(Y,Z).         /*  parent of Z.                       */

                        et cetera
```

Table 4.4. Example of **predecessor_2/2** using non-recursive programming [4:15]

In general, the meaning of a PROLOG rule should be very easy to understand, as long as the programmer obeys a general set of style guidelines. Functors should have meaningful names; the body of the rule should be indented with respect to the head; only one functor should be written on a line; comments (delimited by the character pairs /* and */, or all text from the character % to the end of a line) should be generously used throughout the program; and comments written in the body of a rule should be aligned to the right of the terms of the clause. When possible, all clauses of a rule should be located together, and all singleton variables (those whose names only occur once in a clause) should have names that begin with an underscore. Some simple PROLOG procedures that demonstrate these guidelines are shown in Table 4.3, along with comments explaining their meaning.

The functors **predecessor/2** and **gcd/3** in Table 4.3 demonstrate PROLOG's ability to use recursion in order to describe their relationships. By using recursion, the functor **prececessor/2** can be used to find ancestral predecessors at any depth. If one were to attempt a definition of **predecessor/2** using explicit definitions, the clauses in Table 4.4 might result [4:15]. However, it is quickly seen that this new definition of **predecessor_2/2** will require a new clause for every level of the predecessor relationship. Since a parent

Formal Object	"Cons pair" Notation	Element Syntax
.(a,[ ])	[a|[ ]]	[a]
.(a,.(b,[ ]))	[a|[b|[ ]]]	[a,b]
.(a,.(b,.(c,[ ])))	[a|[b|[c|[ ]]]]	[a,b,c]
.(a,X)	[a|X]	[a|X]
.(a,.(b,X))	[a|[b|X]]	[a,b|X]

Table 4.5. Equivalent forms of lists [31:44]

is a predecessor, it requires a clause. Since grandparent, great-grandparent, and great-great-grandparent are also predecessors, they receive their own clauses, as well. Using this definitional technique, additional clauses must be added until the functor reaches the depth of a given family tree. If the family tree changes, the definition of `predecessor_2/2` (and perhaps other functors) may have to change, as well. The recursive definition of `predecessor/2` in Table 4.3, however, applies to family trees of any depth. This fact, combined with the simplicity of its definition, makes this definition preferable to the explicit definition in Table 4.4. Additionally, many relationships (such as greatest common divisor) are most easily described in terms of recursion, yielding such expressions as `gcd/3` in Table 4.3.

4.2.3 PROLOG *and Lists.* Lists play an important role in PROLOG and receive special handling as a result. A list is a binary structure whose first argument, or *head*, holds an *element* and whose second argument, or *tail*, contains the remainder of the list [31:43]. In a proper list, the head can contain any PROLOG object as an element, but the tail should contain another list [4:68]. It is obvious from this definition that lists are recursively defined structures. The base case of this recursive definition is the *empty list*, otherwise known as *nil*, represented by the symbol [ ]. PROLOG constructs lists by using the functor ./2 (pronounced as "dot"), but a second syntax is provided for ease of use. Thus, the list .(X,Y) is usually written as [X|Y], where X is called the *head* of the list and Y is called the *tail* of the list. Since, in a proper list, Y would itself be a list, those readers familiar with Lisp will recognize that this representation of a list provides an analog to the Lisp 'cons' function, with Lisp using the notation `(X.Y)` as the equivalent of PROLOG's `.(X,Y)`. Additionally, the final empty list is usually omitted from the written notation for the list [31:43]. Examples of the various equivalent notations for lists are shown in Table 4.5. It should be noted that by using the dot functor directly, rather than using the alternate element-syntax, one can construct terms that are more general than lists [31:43]. While

```
member(X,[X|_Xs]).          /* Element X is a member of a list   */
                            /*  that begins with element X.      */
member(X,[_Y,Ys]) :-        /* Element X is a member of a list   */
   member(X,Ys).            /*  [_Y,Ys] if it is a member of the */
                            /*  tail Ys of that list.            */

length([],0).               /* The empty list has length 0.      */
length([_X|Xs],L+1) :-      /* Any other list has a length that  */
   length(Xs,L).            /*  is one greater than the length   */
                            /*  of its tail.                     */

reverse([],[]).             /* The reverse of an empty list is   */
                            /*  itself an empty list.            */
reverse(List,Reverse) :-    /* The reverse of a non-empty list   */
   reverse(List,[],Reverse).
                            /*  is found by using reverse/3.     */

reverse([X|Xs],Acc,Ys) :-   /* Reverse the list [X|Xs] by moving */
   reverse(Xs,[X|Acc],Ys). /*  X to the front of accumulator    */
                            /*  Acc and reversing the tail Xs.   */
reverse([],Ys,Ys).          /* If the list to be reversed is     */
                            /*  empty, then the accumulator Acc  */
                            /*  contains the reversed list.      */
```

Table 4.6. Example PROLOG for **member/2**, **reverse/2** [31:45,48] and **length/2** [4:92-93]

the term

```
.(a,.(b,c))                                    (4.10)
```

is a legal use of the ./2 functor, its result

```
[a,b|c]                                        (4.11)
```

is not a proper list.

Since lists are recursively defined structures, it is natural that many list-processing operations are themselves recursive. The use of recursion allows the functor to "move along" the list in order to perform some operation upon the list. This is demonstrated

in Table 4.6, where `member/2`, `reverse/2`, and `length/2` are shown, as well as auxiliary predicate `reverse/3`.

4.3 Quintus PROLOG

Quintus PROLOG is an Edinburgh-compatible PROLOG which is commercially available for VAX/VMS, UNIX, and other operating systems. (A version of Quintus PROLOG (in reality LPA-PROLOG) is available for MS-DOS, but is not compatible with the VAX/VMS and UNIX versions.) It provides a fast, efficient implementation of PROLOG along with a considerable number of additional features. Quintus includes a program entry and debugging interface to the Emacs text editor (for the VAX/VMS and UNIX versions), special functions to allow interfacing with programs written in other languages, style and syntax warning and error messages, a comprehensive interactive debugger, a substantial library of list processing, term manipulation, and input/output routines, and an on-line documentation and help facility [28:1]. A number of these features are discussed in the subsections below.

4.3.1 Using Emacs In Quintus. Quintus PROLOG is supplied with a version of Emacs written by Unipress Software, Inc. Emacs is a highly customizable editor written in a Lisp-like language. To run PROLOG under the Emacs interface, the user types a command such as:

prolog + (4.12)

or

prolog + *file-to-be-edited* (4.13)

at the UNIX command prompt. This will cause Emacs to be invoked with two windows: PROLOG runs as a subprocess in the lower window, while *file-to-be-edited*, if supplied by the user, is loaded into the upper window [27:21].

By using the Emacs interface, one has access to a number of additional ways of manipulating text in the PROLOG environment. Since PROLOG is run as a subprocess of the Emacs editor, all Emacs editing commands can be used in the PROLOG window. Thus, one can copy all or part of a previous query and then deposit it on the current PROLOG command line, or scroll within the window in order to view previous screens of text [27:21]. By using the command sequence **Control-o**, one can move the focus amongst the various displayed windows. By moving the focus into the editor window, moving the

cursor onto a given PROLOG procedure, and then pressing **Escape i**, that procedure will the loaded into the PROLOG interpreter. The command **Control-x.** will locate the source code associated with any given functor name and arity, loading the source code into the editor window [27:23]. To exit the Emacs environment, the command **Escape Control-c** should be used. Other commands of interest may be found in the Quintus PROLOG System Dependent Features Manual, Chapter 4, and the Quintus Prolog User's Manual, Chapter 4 and Appendix III.

4.3.2 Style and Syntax Restrictions. Quintus PROLOG imposes a number of style and syntax restrictions upon what is technically proper PROLOG in order to make its operation more straightforward. There are nine main suggestions for program layout which are given by Quintus, shown below. While the use of all nine are recommended, items 1, 2, 3, and 4 are necessary in order to fully utilize the Emacs interface [28:65-66].

1. Group PROLOG clauses of the same name and arity together in one location.
2. Start the heads of all PROLOG clauses at the beginning of a line, indenting all additional lines for the clauses.
3. If a comment line continues onto a following line, indent the continuation line.
4. Do not write clause definitions that use operators in the heads of the clauses.
5. Use blank lines between procedures, but not between clauses of the same procedure.
6. Write each goal on its own line.
7. Use a comment of the form /* *text* */ immediately above each procedure in order to detail any assumptions about the arguments and to explain what actions the procedure performs.
8. Use comments of the form % *text* to the right of goals in the body of a procedure, avoiding the use of /* *text* */ comments on lines of code.
9. When possible, try to use meaningful variable names.

In addition to the layout guidelines given above, Quintus imposes three style conventions upon user programs.

1. Define all clauses for a given procedure in one file, as the consult/1 and compile/1 functors do not allow the definition of a procedure to be spread across more than one file. (This limitation can be circumvented if the procedure in question is declared as a multifile or dynamic procedure, as discussed in Section 4.3.3.)

```
check_state(TheState) :-
    old_state(TheStaye,X), %  Error occurs on this line
    write(TheState),
    write(X).
```

Table 4.7. Misspelled variable resulting in singleton variable [28:49]

2. All clauses for a given procedure should be contiguous in the source file.
3. If a variable appears only once in a clause (otherwise known as a *singleton variable*), the name for the variable should begin with the character '_'.

If any of these conditions is not met, Quintus PROLOG will produce a warning message when the file is consulted. If style convention 1 is violated, PROLOG will ask whether the new definition should replace the existing procedure definition, or if the new definition should be ignored instead. If style convention 2 is violated, PROLOG produces a warning message of the form:

[WARNING: Clauses for foo/2 are not together in the source file]

Similarly, if style convention 3 is violated, such as in Table 4.7, PROLOG will issue the warning message:

[WARNING: Singleton variables, clause 1 of check_state/1: TheStaye]

If desired, some or all of these style warning facilities can be toggled off and on by using the **no_style_check/1** and **style_check/1** predicates, using the arguments **all**, **single_var**, **discontiguous**, or **multiple**, as appropriate [28:48-49].

The Quintus manuals mention that the use of disjunctions is usually unnecessary. However, if they are to be used, Quintus recommends the use of the '|' symbol in place of the standard ';' (semicolon) in order to promote clarity of the resultant code. Table 4.8 shows an example of the use of the '|' symbol. It is important to note, however, that since the semicolon has been historically used as the disjunction symbol, the Quintus system will automatically translate the '|' symbol into a semicolon for its internal representation

```
bank_open(Day,Time) :-
    weekday(Day),           % The bank is open on weekdays
    \+ bank_holiday(Day),   %  except bank holidays
    1000 =< Time,           %  from 10 a.m.
    (  Time =< 1500,        %  until 3 p.m.
       \+ friday(Day)       %  Monday through Thurday
    |  Time =< 1800,        %  or 6 p.m.
       friday(Day)          %  on Fridays
    ).
```

Table 4.8. Example of the use of | in disjunctions [28:67]

of the program. Thus, any printouts that are generated by the Quintus system will show semicolons, even if vertical bars were present in the original source code files [28:67].

4.3.3 Control and Directive Constructs. Quintus PROLOG is equipped with a number of control constructs, most of which are shown in Table 4.9 [26:45-49]. The use of control constructs in program code provides the programmer with a means of specifying "how" the logic program is executed, rather than simply specifying "what" is to be accomplished. Thus, while the use of control constructs is necessary for the construction of real-world PROLOG programs, they should be used sparingly and only where necessary. Use of control constructs, when not carefully considered, can easily change the meaning of a PROLOG program without any intent to do so.

Quintus PROLOG allows the use of directives within source files. A *directive* is a query that is located inside a PROLOG source file, rather than being supplied by the user at the PROLOG command line. Directives are written as terms with principal functors `:-/1` or `?-/1`, and are executed as they are encountered. Quintus PROLOG treats the `:-/1` and `?-/1` functors equivalently, but recommends the use of the `:-/1` functor in source files in order to enhance program clarity. Any legal PROLOG clause may be used as a directive. One common type of directive to have in a file causes the consultation of a second file, such as:

`:- consult(` *subfilename*`).` (4.14)

It should be noted that debugging will not be enabled during the execution of a directive, regardless of whether top-level debugging has previously been enabled. A directive such as

`:- trace, do_verification.` (4.15)

can be used to overcome this limitation through an explicit call to either **debug/0** or **trace/0** [26:34].

Two additional Quintus PROLOG declarations are important in the construction of the **AFIT_VERIFY** program. The first, **multifile/1**, is used to declare that a predicate is defined across a number of separate files. This directive is of the form

```
:- multifile PredicateSpecification                    (4.16)
```

where *PredicateSpecification* is one or more predicate specifications (name and arity) separated by commas. For example,

```
:- multifile module_name/1, port/4, part/3, state_eqn/2.    (4.17)
```

would declare that each of these predicates is spread across more than one file. This declaration should be placed in the first file of the sequence of files containing these predicate specifications [26:42]. Similarly, **dynamic/1** is used to allow new clauses for a specified predicate to be dynamically inserted into the database (using **assert/1**) or removed from the database (using **retract/1**). The syntax for **dynamic/1** is identical to that shown for **multifile/1** in (4.17) [27:81-82].

4.3.4 Library Routines. Quintus PROLOG includes a set of supplemental files that are organized into an online library structure. The library contains a large number of predicates which, although not native to the PROLOG environment, can be regarded as extensions to the PROLOG system. By default, the predicate **library_directory/1** has a clause for the library directory, and this predicate can be altered by the user in order to add additional directories to the library search path. The definition of **library_directory/1** is utilized by the PROLOG system in order to allow the user to refer to any library file by using the syntax **library(***FileName***)**. For example,

```
:- compile(library(lists)).                            (4.18)
```

would load the library file **lists.pl**, along with any files it depends upon, into the database [25:1].

There are a number of different library files, each concerned with a different area of interest. There are four packages (`basics.pl`, `lists.pl`, `sets.pl`, and `ordsets.pl`) that contain list-oriented operations [25:18], six packages (`arg.pl`, `change_arg.pl`, `occurs.pl`, `same_functor.pl`, `subsumes.pl`, and `unify.pl`) that extend PROLOG's built-in set of operations on terms [25:36], one package (`strings.pl`) concerned with text processing [25:47], eight packages (`lineio.pl`, `continued.pl`, `ask.pl`, `prompt.pl`, `read_in.pl`, `ctypes.pl`, `read_const.pl`, and `read_sent.pl`) that deal with obtaining user input [25:93-110], one package (`not.pl`) concerned with various types of inequalities [25:71], eight packages (`files.pl`, `ar_open.pl`, `ask.pl`, `big_text.pl`, `crypt.pl`, `directory.pl`, `unix.pl`, and `fromonto.pl`) concerned with various file operations [25:77-85], as well as some unsupported library packages [25:111]. Many of these PROLOG library files actually load compiled, optimized code which rapidly executes functions that would be much slower if actually supplied in PROLOG. Through the use of these prewritten library packages, the **AFIT_VERIFY** program has been enhanced through an upgraded user interface and the introduction of a component library system.

4.4 Summary

As shown throughout this chapter, PROLOG provides a rich environment for programming. Its pattern-matching abilities make it uniquely suited to logic programming applications, such as the VERIFY and **AFIT_VERIFY** systems. PROLOG contains a full set of control structures, as well as a unification-driven execution model, and can be shown to be a Turing-complete language [31:228]. A summary of some of the more common PROLOG procedures and control constructs is shown in Table 4.10 [22:64].

P, Q Conjunction: (P, Q) succeeds if P succeeds and then Q succeeds.

P; Q Disjunction: (P; Q) succeeds if P succeeds or Q succeeds. The character '|' may be used as an alternative to ';'.

! Cut: When first encountered as a goal, cut always succeeds. If backtracking should cause PROLOG to return to the cut, the current clause will fail.

call(X) If X is instantiated to a term, then the goal **call(X)** is executed as if **X** actually appeared in its place. However, when **call(X)** is executed, any cuts in X will only cut alternatives in the execution of X, not in the clause in which it occurs.

\+ P This fails if P has a solution and succeeds otherwise, with the condition that P may not contain a cut. Thus, **\+ P** is equivalent in behavior to **(P -> fail ; true)**.

P -> Q ; R Conditional goal: This statement is read "if P then Q, else R" and selects between the execution of Q and R, based upon whether P succeeds. It should be noted that if P succeeds and Q fails, then backtracking into P does not occur. Additionally, P may not itself contain a cut. The predicate ->/2, also known as 'local cut', acts as if it were a cut whose range is restricted to within the disjunction, as it cuts away R and any choices within P. The precedence of ->/2 and ';' are 1200 and 1100, respectively, while the precedence of ',' is 1000, so the statement **P, Q -> R, S; T** is equivalent to **((P, Q) -> (R, S)) ; T**.

P -> Q This is equivalent to the statement **P -> Q ; fail**.

true This statement always succeeds.

otherwise This statement, like true/0, always succeeds. The predicate otherwise/0 is often used for constructing conditionals.

fail This statement always fails.

false This statement, like fail/0, always fails.

repeat Predicate repeat/0 always succeeds, whether entered directly or by backtracking, and is generally used to simulate looping constructs found in procedural languages. Use of repeat/0 is not recommended, as recursion can produce similar results.

Table 4.9. Control Constructs in Quintus PROLOG [26:45-49]

PROLOG Notation	Meaning	Example
,	*AND*: Satisfied if both terms before and after the *AND* are satisfied.	`child(X,Y),male(Y)`
:-	*IF*: The head of the rule is true *IF* the body of the rule is true.	`parent(X,Y) :- child(Y,X).`
;	*OR*: Satisfied if either the term before or after is satisfied.	`file_exists(F) ; fail`
[ ]	Square brackets enclose a list.	`[nand,nor,xor]`
|	(a) Denotes an element at the head of a list. (b) Equivalent to ; in Quintus PROLOG.	If `[1,2,3] = [H|T]` then `H = 1, T = [2,3]` `file_exists(F) | fail`
!	*CUT*: Within the current procedure, don't backtrack through the *CUT*.	`abs(X,Y) :- X<0,!,Y is -X. abs(X,X).` *CUT* keeps abs/2 single-valued.
=	Unifies terms on both sides.	If `f(X,b,Z)=f(a,Y,Z)`, then both sides become `f(a,b,Z)` and `X=a, Y=b`
=..	*UNIV*: Converts between lists and terms.	`fact(N,Factorial) =.. [fact,N,Factorial]`
assert, asserta	Add a new fact or rule to the 'top' of the PROLOG database	`asserta(flag(loaded,xor))`
assertz	Add a new fact or rule to the	`assertz(day(tuesday))`
retract	Remove one fact or rule from the PROLOG database.	`retract(day(monday))` removes this entry
retractall	Remove all matching facts or rules from the PROLOG database. (Similar to *retract*,but *retractall* always succeeds.)	`retractall(day(_X))` removes all matching entries that might exist in the database
call	Execute the procedure that is the argument of *call*	`call(fact(N,Fact))`
Lowercase Identifiers	Atoms: used for names of procedures or data	`fact, write`
Uppercase Identifiers	Variables	`Date, ModuleName`

Table 4.10. Common PROLOG Procedures and Constructs [22:64]

V. *Program Development of* **AFIT_VERIFY**

5.1 Background

Captain Kevin Sparks originally developed the **AFIT_VERIFY** system to model the behavior of Barrow's VERIFY system [30:Ch 5, 1]. **AFIT_VERIFY** was originally written using PROLOG-1 under MS-DOS. As Captain Sparks continued to develop the program, however, the limitations that resulted from the small memory model used in PROLOG-1 forced him to rehost the program under Quintus PROLOG [30:Ch 5,3]. This adaptation of the PROLOG-1 version of **AFIT_VERIFY** to Quintus PROLOG was performed in the most direct and rapid method possible, consisting mainly of changing the precedence of several operands and defining some predicates as being stored in multiple files (as opposed to a predicate's definition being loaded from a single file). Despite the fact that the program was still operating under a PROLOG-1-oriented design philosophy, Sparks's final version of **AFIT_VERIFY** provided between a 5X and 15X speedup when compared to the final PROLOG-1 version [30:Ch 5, 2].

5.2 Initial Development Efforts

5.2.1 Analysis of Sparks's Final **AFIT_VERIFY**. Code development work for the current thesis began with an analysis of Sparks's final version of **AFIT_VERIFY**. Examination of the code revealed that the system was designed around the PROLOG-1 environment, with the addition of the minimal number of modifications necessary for the program to operate in the Quintus PROLOG environment once the **AFIT_VERIFY** program exceeded the limited stack space provided by PROLOG-1. It was especially evident that the program did not provide a user-interface, other than that provided by the PROLOG environment itself.

Sparks's **AFIT_VERIFY** provides a command-line–style interface to the user. At the PROLOG prompt, one must first load the **AFIT_VERIFY** program into memory, followed by loading the various files needed to define the system under test. The execution of the **AFIT_VERIFY** program is then invoked through the `verify/1` procedure, specifying the name of the top-level module as the argument.

The `verify/1` procedure was broken into five clauses in order to cover the five possible types of modules: previously verified modules, primitive modules with no state information, primitive modules with state information, non-primitive modules without

```
verify(Module) :-                          /* previously verified module   */
   verified(Module),
   !,
   writeln(['>>>',Module,' previously verified >>>']).
verify(Module) :-                        /* primitive module with no state  */
   not part(Module,_,_),
   not state_eqn(Module,_),
   !,
   asserta(verified(Module)),
   writeln(['>>>',Module,' primitive (needs no verification)>>>']).
verify(Module) :-                          /* primitive module  with state  */
   not part(Module,_,_),
   !,
   asserta(verified(Module)),
   writeln(['>>>',Module,' primitive (needs no verification)>>>']).
verify(Module) :-                          /* non-primitive with no state  */
   not state_eqn(Module,_),
   writeln(['>>> Attempting to verify ',Module,'>>>']),
   verify_components(Module),
   !,
   derive_and_equate_behaviors(Module),
   asserta(verified(Module)),
   writeln(['<<< Success! Behavior of',Module,'meets its specification.']).
verify(Module) :-                              /* non-primitive with state  */
   writeln(['>>> Attempting to verify ',Module,'>>>']),
   verify_components(Module),
   !,
   derive_and_equate_behaviors(Module),
   derive_and_equate_states(Module),
   asserta(verified(Module)),
   writeln(['<<< Success! Behavior of',Module,'meets its specification.']).
```

Table 5.1. Sparks's Implementation of **verify/1**

state information, and non-primitive modules with state information. These five clauses are shown in Table 5.1. This delineation proved to be effective in directing the verification work to be performed and has been retained in the updated system.

As indicated in Table 5.1, the `verify/1` clauses invoke a number of other procedures. The procedures `derive_and_equate_behaviors/1`, `derive_and_equate_state/1`, and `verify_components/1`, as their names indicate, allow the `verify/1` clauses to hierarchically verify each component of a given module, ensuring that all outputs and states are equivalent. The `verify_components/1` procedure implements a fail-driven loop which, for each `part(Module,Name,Component)` fact in a given module, methodically invokes the `verify/1` procedure in turn on each component in the module.

Digital circuits are defined in Captain Sparks's implementation by a set of facts, similar to those used by Barrow [2]. In summary, a given module, `ModuleName`, is defined by its input and output ports

```
port(ModuleName,PortName,InputOrOutput,SignalType)
```

its list of subcomponents

```
part(ModuleName,LocalName,SubModuleName)
```

a wire list of internal connections between the input and output ports of the module and its subcomponents

```
connected(ModuleName,SourcePort,DestinationPort)
```

and the specified output behavior of the module

```
output_eqn(ModuleName, OutputPort := SpecifiedBehaviorFunction) .
```

If a module is primitive, then it contains no subcomponent modules and its specified behavior is defined to be equivalent to its derived behavior. If a module has state variables, then it will have a number of clauses defining the state specification: a definition of each

state variable that is available to the external environment

```
state_of(ModuleName,ExternalStateName,SignalType)
```

a listing of the states internal to the module's subcomponents that actually produce the state information

```
state_map(ModuleName,ExternalStateName,InternalStateName)
```

and a simple definition of the state transitions for each given state

```
state_eqn(ModuleName,ExternalStateName := NextStateFunction) .
```

It is important to note that only those state equations that follow a regular set of transitions (such as "increment by one") can be easily described. Although an IF/THEN/ELSE construct is available for use, describing the transitions of a complex automaton can rapidly become unmanageable.

Captain Sparks made use of the `setof/3` procedure when he implemented both the `derive_and_equate_state/1` and `derive_and_equate_behaviors/1` procedures. The `setof(Templace,Goal,SetName)` procedure is invoked with three arguments: a `Template`, which provides the form for the elements in the resulting list, a `Goal`, and a `SetName`, which returns the resulting set (expressed as a list) of all instances of `Template` such that `Goal` is satisfied. As an example, the `derive_and_equate_behaviors/1` procedure included the statement

```
setof(Outputs,output_eqn(Module,Outputs := _),Outlist)        (5.1)
```

which should produce a list, `Outlist`, which contains a number of `output_eqn/2` elements. Since `Module` is instantiated when this procedure is invoked, only those `output_eqn/2` facts in working memory that pertain to the given module should be in the list.

Since PROLOG-1 did not provide the `setof/3` procedure, Captain Sparks wrote his own `setof/3`. When he moved **AFIT_VERIFY** from PROLOG-1 to Quintus PROLOG,

however, he discovered that Quintus provides a **setof/3** procedure as a library function. The Quintus library procedure, however, did not behave precisely in the same fashion as Captain Sparks's procedure. Due to the fashion of the differences, along with the particular test circuits used in Captain Sparks's research, this did not cause any difficulties during his thesis work. When subjected to larger circuits with more outputs and states, however, this code no longer produced correct output. In order to produce the proper set, the statement

```
setof(Outputs,Dummy1^output_eqn(Module,Outputs := Dummy1),Outlist)    (5.2)
```

was substituted. In this statement, `Dummy1^output(Module, Outputs := Dummy1)` specifies that the variable **Dummy1** is to be existentially quantified over the goal. As a result, Equation (5.2) produces a single set containing all of the `output_eqn(Module,Outputs := Dummy1)` facts for the given **Module**, while the syntax used in Equation (5.1) produces, upon backtracking, a one set for each `Output,Dummy1` pair. This problem, in fact, only manifested itself on both such **setof/2** procedure calls in **derive_and_equate_behaviors/1**.

Another problem was discovered in the **derive_and_equate_behaviors/1** procedure with respect to **setof/2**. In this case, the statement

```
setof(Outputs,Dummy2^derived_behavior(Module,Outputs := Dummy2),Derlist)
```

returned a "set", **Derlist**, in which a number of the derived behavioral outputs were *almost* identical, except in one respect – the name of their uninstantiated variable. Thus, **Derlist** might be returned as a list such as that shown in Equation (5.3).

```
[sum0(_132),carry(_318),sum0(_113),sum1(_823),carry(_1238)]    (5.3)
```

In order to properly compare these derived states to the states in the module specification, however, this list needs to be "compacted." The procedure **setof_to_trueset/2**, along with helper procedure **unifiable_with_list/2**, was written to "compact" the list in Equation (5.3) into that shown in Equation (5.4).

```
[sum0(_132),carry(_318),sum1(_823)]    (5.4)
```

Finally, although Captain Sparks had both `derive_and_equate_behaviors/1` and `derive_and_equate_states/1` perform an arithmetic comparison to ensure that the number of specified outputs (or states) was the same as the number of derived outputs (or states), this was not a sufficient test. It is also necessary to test that these lists contain elements which are unifiable. As an example, given the sample lists of derived and specified outputs

```
[sum0(_132),carry(_318),sum1(_823)]
[carry(_123),sum0(_124),sum1(_125)]
```

we can unify `sum0(_132)` with `sum0(_124)`, `sum1(_823)` with `sum1(_125)`, and `carry(_318)` with `carry(_123)`. This would be an acceptable match between the derived and specified outputs. If, however, the verification process were to produce the specified list

```
[carry(_7770)]
```

and the derived list

```
[sum(_7878)]
```

(as, in fact, occured at one point during the testing of the `halfadd` component) then **AFIT_VERIFY** should not accept these lists as matching, even though the number of elements is identical. The procedure `unifiable_lists/2` was written to handle this problem, and is shown in Appendix A.1.1.

The majority of the work in the **AFIT_VERIFY** system is performed by the `derive_behaviors/3`, `equal_behaviors/3`, `derive_states/3`, and `equal_states/3` procedures. `Derive_behaviors/3`, through the helper procedure `derive_behavior/3`), works its way from each output back toward the inputs, replacing the components found along the way with their specified behaviors (after first recursing to verify that their specified behaviors meet their implementation-derived behavior). Any Boolean or mathematical equations that are encountered are expanded and canonicalized using the `evaluate1/2` procedure, and any other type of behavior is left unchanged. If new behaviors were to be added to the **AFIT_VERIFY** environment, additional `derive_behavior/3` clauses would be required.

The `evaluate1/2` procedure performs a simple canonicalization of its first argument, returning the result as its second argument. The `evaluate1/2` procedure uses the `evaluate_brown/2` procedure to actually perform the canonicalization, then displaying the result to the user before returning to the calling procedure. The true work of canonicalizing the behavioral expression is done by the `evaluate_brown/2` clauses. Captain Sparks, as guided by Dr F.M. Brown, implemented the evaluation of AND, OR, NEG (logical negation), IF/THEN/ELSE, and '+' (addition) operations. As new behaviors are added to **AFIT_VERIFY**, it is clear that new `evaluate_brown/2` clauses would need to be added as well. This code has already been expanded to allow for evaluation of behaviors that are specified as NAND or NOR logic.

Both the `derive_states/3` and `derive_behaviors/3` procedures make use of the `substitute_state/3` procedure. `Substitute_state(Module,OldBehavior,NewBehavior)` is used in `derive_states/3` to derive the next state in a module with state equations, or in `derive_behaviors/3` to substitute a derived behavior into the previously derived state equations. `Substitute_state/3` uses the `state_map/3` facts which define the connection of internal and external states in a module, along with the `replace_all/5` procedure which, given the `state_map/3` information, replaces the internal state variables in a submodule's derived behavior with the appropriate externally-visible state variables, thereby deriving a new derived behavior for use in the enclosing module.

5.2.2 Attempted Integration of Scheme with PROLOG. As previously mentioned in Section 1.4, one objective of this thesis was to allow for the incorporation of additional proof strategies into the **AFIT_VERIFY** framework. Although PROLOG is a Turing-complete language, it can be a difficult language for writing elaborate functional procedures Since the Scheme language (a variant of Lisp) is well suited to the implementation of functional procedures, a copy of Scheme Prolog 1.1, written by John Cleary *et al.* of the University of Calgary was obtained. This package provides a fairly simple interpreter for pure PROLOG, implemented in the Scheme language. It was hoped that, by integrating Captain Sparks's **AFIT_VERIFY** into a Scheme-based environment, that the addition of new proof strategy function modules would be simplified. In order to support the Scheme Prolog 1.1 package, C-Scheme 6.1 was obtained from the Massachusetts Institute of Technology (MIT) and temporarily installed on a local UNIX-based computer system (galaxy.afit.af.mil). Approximately two weeks were spent obtaining the source code for both of these software packages, installing them on a local computer, and modifying Scheme Prolog 1.1 for compatibility with C-Scheme 6.1 syntax. Once this initial task of

```
?- op(100, fy, not).

not X :-
        X,
        !,
        fail
        ;
        true.
```

Table 5.2. Sparks's implementation of **not/1** for Quintus PROLOG

producing an executable version of Scheme Prolog 1.1 was completed, however, it was discovered that the combination of Scheme Prolog 1.1, C-Scheme 6.1, and **AFIT_VERIFY** was not completely compatible. Major causes of this incompatibility were the distribution of the **AFIT_VERIFY** code throughout multiple files and the extensive use of control structures (such as cut) within the **AFIT_VERIFY** program. This path of investigation was therefore terminated due to the projected number of hours that would have been necessary to integrate these packages. Since these tools were not absolutely essential to this thesis, efforts were shifted instead into integrating the **AFIT_VERIFY** program with the Quintus PROLOG environment.

In order to prove that the derived behavior is equivalent to the specified behavior, Sparks's **AFIT_VERIFY** system applies a small (but powerful) set of rewriting rules. Much of the work in this area is performed by the Boolean expansion code originally created by CPT Michael A. Dukes [12]. Other simple rewrite rules, such as rewriting `IntegerVariable + 1` as the mathematically equivalent `1 + IntegerVariable`, are also used by the system. Additional transformation rules can be added as alternative clauses in the `equal_behaviors/3` and `equal_states/3` procedures.

5.3 Initial Integration of **AFIT_VERIFY** *into Quintus* PROLOG

Kevin Sparks's version of AFIT_VERIFY was designed to be compatible with the PROLOG-1 environment and the MS-DOS operating system. Quintus PROLOG uses a default file type of `.pl`, while PROLOG-1 uses a default of `.pro`, so all source file names were converted accordingly. Since MS-DOS uses a carriage return (CR) and line feed (LF) pair to mark an end-of-line, while UNIX uses only a line feed, all extraneous carriage return characters were immediately removed from the source files.

```
:- op(900, fy, not).

not Goal :-
          \+ call(Goal).
```

Table 5.3. Improved implementation of **not/1** for Quintus PROLOG

```
:- op(900, fy, not).

not Goal :-
    free_variables(Goal, [], [], Vars),
    Vars = [_Any1|_Any2],
    !,
    error_break('~N! free variables ~p~n! in goal not(~p)~n',
                [Vars,Goal]),
    \+ call(Goal).              % Act like \+, if user agrees

not Goal :-
    \+ call(Goal).
```

Table 5.4. Implementation of pseudo-logical negation for Quintus PROLOG

While PROLOG-1 supplies the `not/1` predicate, Quintus PROLOG does not provide this functor. Sparks provided a minimal implementation of `not/1` for Quintus PROLOG, shown in Table 5.2, but this implementation did not mesh smoothly with the Quintus environment. In particular, PROLOG-1 uses operators whose precedence ranges from 0 to 255, while Quintus PROLOG uses a range from 0 to 1200. The implementation of `not/1` shown in Table 5.3 not only provides this operator with an appropriate precedence, but also uses the Quintus \+ operator. If `call(Goal)` can succeed, then `not Goal` will fail, but if `call(Goal)` does not have a solution, then `not Goal` will succeed.

It should be noted that this implementation of `not/1` provides an "is not provable" operator, as opposed to the "is not true" operator of formal logic. An implementation of `not/1` that corresponds more closely to logical negation, shown in Table 5.4, must first ensure that `Goal` is completely defined (does not have any non-ground instances) before executing the `\+call(Goal)` operation. Quintus provides the library func-

tor `free_variables/4`, invoked as

```
free_variables(Generator, Template, OldList, NewList)
```

where NewList contains all universally quantified variables (i.e., those yet unbound) in Generator, less those which occur in Template, with OldList used as an accumulator. Thus, in Table 5.4 all unbound variables in Goal are moved to list Vars, which is then tested to be non-empty. If it is non-empty, `error_break/2` writes an error message to the output, where the first argument is a formatting string and the second argument contains a list of variables or constants to be applied to the formatting string. If the debugging break level is set appropriately, Quintus will then continue by executing `\+ call(Goal)` regardless of the error condition; otherwise the procedure will fail with the cut preventing the execution of the second clause. If the list Vars is empty, however, this indicates that Goal consists only of ground instances. The first clause will fail and the second clause will be executed instead. An implementation of this variety, however, introduces additional processing overhead and is not necessary for the task at hand. If necessary, however, it could easily replace the currently used definition in Table 5.3.

So as to allow the redefinition of the `module_name/1`, `port/4`, `part/3`, `state_of/3`, `state_eqn/2`, `state_map/3`, `output_eqn/2`, and `connected/3` predicates by the various components being verified, the `multifile/1` and `dynamic/1` directives are applied to each of these predicates. The use of `multifile/1` declares that the clauses of the procedure may be found in more than one file, and `dynamic/1` declares that clauses may be asserted and retracted during program execution.

5.4 Enhancements to the AFIT_VERIFY *System*

Sparks's **AFIT_VERIFY** system provided a good, minimal emulation of Barrow's VERIFY environment. When using Sparks's program, the user loads the necessary routines into the PROLOG environment (usually through a specially constructed PROLOG source file) and then manually invokes the verifier upon the component in question.

A substantial effort was put into enhancing the user interface to the **AFIT_VERIFY** system, taking full advantage of the library routines provided by Quintus and previously discussed in Section 4.3.4. In particular, the `ask_oneof/3`, `prompted_constant/2` and `yesno/2` predicates were used in forming a menu-based user interface while simultaneously guiding the flow of program execution according to the user's choices. This user

```
Welcome to AFIT_VERIFY!
=======================

(Type ? at any prompt if you require help)


Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt:
```

Table 5.5. Opening Screen in **AFIT_VERIFY** System

interface is initially invoked by a directive in the file `qverify.pl` which executes the `do_verify/0` procedure found in `verify.pl`. The top level menu itself is contained in the `main_menu/0` procedure found in the same file. Additional interface procedures, such as `conditional_halt/0`, are also found in `qverify.pl`.

As a result of the extensive work on this user interface, users of the system may now work within the **AFIT_VERIFY** framework without any knowledge of how it is constructed or internally operates. When entering the system, the user is presented with the screen shown in Table 5.5. This menu, like many of the others used in the system, will only accept those choices shown on the bottom prompt (`preload`, `reverify`, `list`, `verify`, `insert`, and `exit`), or enough characters to uniquely identify the menu choice. (Since, in this case, all of the menu choices begin with different letters, the selection can be made with only the first letter being entered.)

Other menus are presented at various times during the session, such as when selecting a library file to be reverified. The user is prompted to select the name of a library component, such as `xor` or `counter`. Once again, the user must select one of the presented

choices and is only required to enter enough characters to uniquely specify the desired part. The menu of library parts is maintained in the file `modfiles.list` as an open list. The **AFIT_VERIFY** system reads in this open list and converts it to a text representation of the list in order for it to be used in the prompting routines. When a part is inserted into the library (using the `insert` option), the system modifies the list of known library files and stores the new list in the `modfiles.list` file.

In order to make **AFIT_VERIFY** into a practical system, it was necessary to allow more than one part to undergo verification during a session. Sparks's implementation of **AFIT_VERIFY** did not allow this to occur, as Quintus PROLOG does not, by default, allow "multifile" clauses in a particular file to be reconsulted during program execution. In addition, Sparks's system did not provide for any "memory" from one verification run during a session to the next verification run. In order to solve these problems, a number of experiments were conducted. It was discovered that the `multifile/1` directive needed to be in the first file of the sequence of different files containing the "multifile" clause. When the `multifile/1` directive is encountered, as in

```
:- multifile part/3.
```

it removes all existing clauses for `part/3` from the database and gives the "new" `part/3` clause the multifile property. Clauses for `part/3` which are subsequently loaded from other files are then added at the bottom of the PROLOG database. If one later attempts to reconsult one of the other files that contain `part/3` clauses, however, Quintus PROLOG prints an error message and refuses to reconsult the clauses.

This refusal caused a number of complications in the development and execution of the **AFIT_VERIFY** environment. If one were to reverify an `xor`, the system would read the clauses in the file `xor.pl` into the database, and then perform the verification. If one were then to verify a full adder that used `xor` as a submodule, such as the part `faddxor`, the **AFIT_VERIFY** system would encounter an error condition when trying to reconsult `xor.pl` as a submodule of `faddxor.pl`. This problem was resolved by moving the task of consulting the module source files into a special source code file, `multdyn.pl`.

The file `multdyn.pl` is stored in the parts library directory and performs three functions. First, it contains the `multifile/1` and `dynamic/1` directives for the `module_name/1`, `port/4`, `part/3`, `output_eqn/2`, `state_eqn/2`, `state_map/3`, `state_of/3`, and `connected/3`

```
load_in(FileName) :-
    ( not flag(loaded,FileName) ->
       (reconsult(library(FileName)),       % load one time, then
         asserta(flag(loaded,FileName))))  %  flag it
    |
    true.                                   % component already loaded
```

Table 5.6. Source Listing for **load_in/1**

predicates used in defining the hierarchical structure and behavior of the digital circuit. Second, it contains the **get_top/1** procedure which actually loads the top level circuit module into the database. (The source code for the file **multdyn.pl** can be found in Appendix A.1.7.) Whenever a digital system is verified, the main program loop first reconsults **multdyn.pl**, thereby redeclaring the **multifile/1** directives. The main circuit file is then consulted into the database via the **get_top/1** procedure in **multdyn.pl**, satisfying the requirement that all **consult/1** commands are executed from the source file containing the **multifile/1** directive.

In conjunction with the use of **get_top/1**, a new set of header directives was added to the digital circuit definition files in order to simplify the hierarchical loading of submodules. The file **qops.pl** contains the **load_in/1** procedure, shown in Table 5.6. The first statements in a circuit definition file should consist of a series of **load_in/1** directives, specifying all of the submodules directly used by the current module. Thus, a set of directives such as

```
:-   load_in(primitive).
:-   load_in(xor).
```

would ensure that the file containing the primitive components (**primitive.pl**) and the file containing the definition of the exclusive-or (**xor.pl**) are both consulted into the database during the loading of the current module. Since **xor.pl** will contain a **load_in/1** directive as well, any submodules necessary to the construction of the **xor** submodule will also be loaded. The procedure **load_in/1** is carefully constructed such that any file from the parts library will only be consulted at most once during the verification of a digital circuit. If a submodule file were consulted more than once, it would in effect "double-wire" a second, identical submodule in parallel with the intended submodule.

```
/*****************************************************/
/*                HALFADD.PL                         */
/*                                                   */
/*****************************************************/


:- load_in(primitive).                    %  get nand2
:- load_in(inv).                          %  get inverter

/*---------------------- halfadd ------------------------*/

:- retractall(module_name(_ModuleNames)).
                        % Make sure that THIS is the topmost
                        %   module up to this point in time
module_name(halfadd).  % Name the current module
```

Table 5.7. Example of PROLOG Directives and Facts In Module Definition File

In addition to using the `load_in/1` directive, a `retractall(module_name(_Name))` directive was added to all module definition files. This directive, which should appear after the `load_in/1` directives but before the `module_name/1` fact, makes certain that the last module name declared (namely, the one for the top-level module being verified) is the only such fact remaining in the PROLOG database. This is essential for proper loading and execution of hierarchically defined modules within the **AFIT_VERIFY** environment. Table 5.7 provides an example of the first few lines of a properly composed module definition file.

5.5 New Modules In **AFIT_VERIFY** *Parts Library*

Captain Mark Mehalic, AFIT/ENG, was consulted for advice on modules that should be added to the **AFIT_VERIFY** parts library. He advised that effort would best be spent in adding parts that were compatible with the Zycad VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) cell library used by AFIT. On the advice of Captain Mehalic, Captain David Banton, Captain Curtis Winstead, and Mr Gene Howell were consulted in order to obtain access to some of the simple circuits that had previously been designed using the Zycad cell library. Since the Zycad cells were based upon two-input NOR gates as well as the two-input NAND gates previously used in

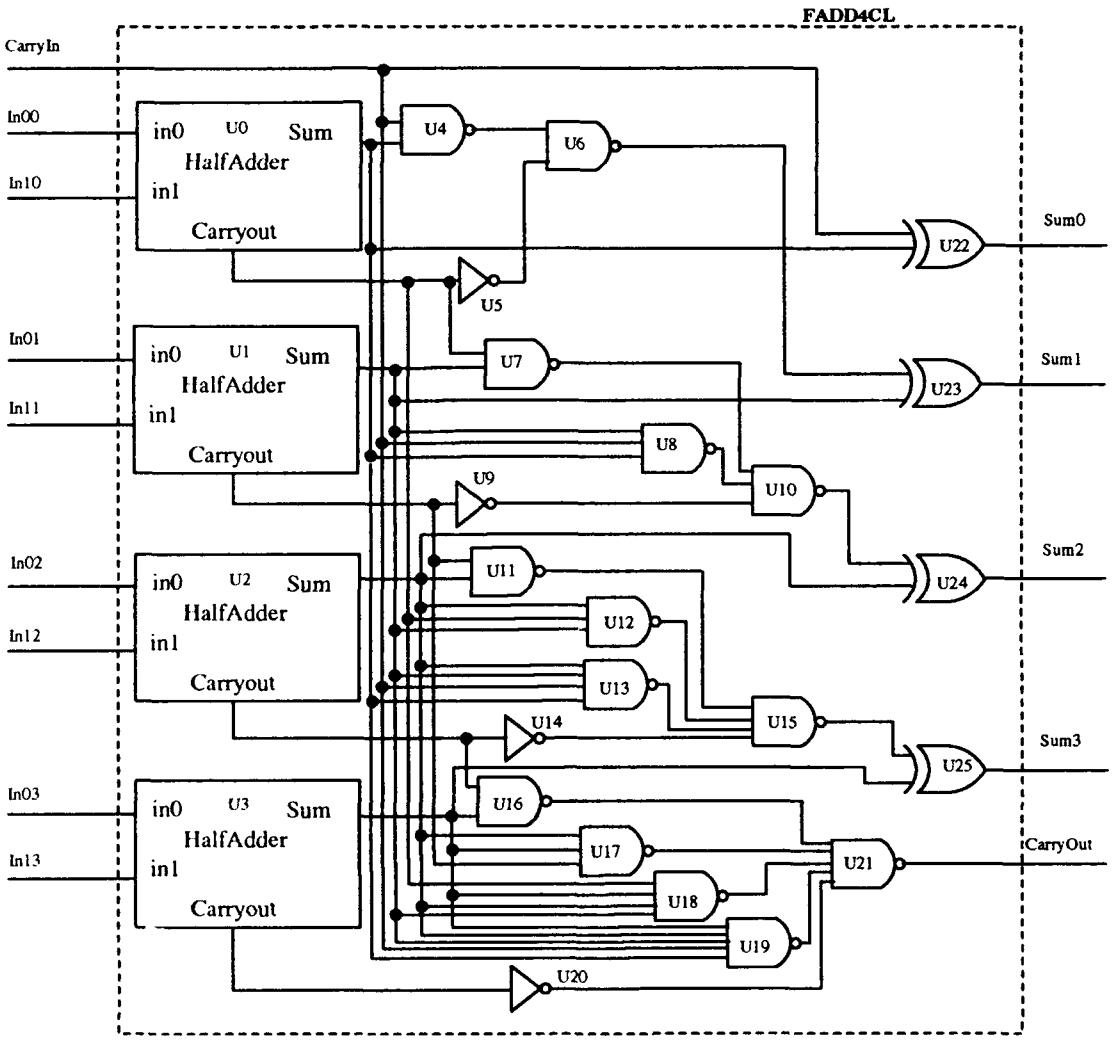


Figure 5.1. Four-Bit Full Adder With Carry Lookahead

AFIT_VERIFY, these students recommended introducing a new primitive component, `nor2`, to the system's set of primitive components stored in the file `primitive.pl`.

In order to perform an accurate translation of the Zycad VHDL parts, it was first necessary to create an inverter module. This file, `inv.pl`, implements an inverter as a two-input NAND gate (`nand2`) whose inputs are connected together. Once an inverter was available, actual Zycad VHDL files were examined for translation into PROLOG syntax. The following files were selected for conversion:

1. An AND-OR-INVERT circuit (`aoi.pl`) based upon an example circuit in the Zycad Reference Manual [32:Ch 10,73].

2. A 4-to-1 multiplexor (`mux_4x1.pl`) composed of 2-to-1 multiplexors (primitive components) and inverters.
3. A half-adder (`halfadd.pl`) composed of two-input NANDs and inverters.
4. A full-adder (`faddnor.pl`) composed of two-input NORs, inverters, and half-adders.
5. A four-bit full adder with carry lookahead (`fadd4_cl.pl`) composed of half-adders, inverters, exclusive ors, and two-, three-, four-, and five-input NANDs.

As mentioned above, the AND-OR-INVERT circuit was based upon VHDL code provided in the Zycad Reference Manual [32:Ch 10,73]. The other circuits were based upon VHDL code written by Captain David Banton, currently a doctoral student at AFIT. Since implementation of these circuits followed similar methodologies, the four-bit full adder will be examined in depth to serve as an example.

5.5.1 Four-Bit Full Adder with Carry Lookahead. In order to implement the four-bit full adder with carry lookahead, it was first necessary to implement a set of three-, four-, and five-input NAND gates. These definitions, found in files `nand3.pl`, `nand4.pl`, and `nand5.pl`, are not highly optimized circuit designs, but instead use combinations of two-input NAND gates (primitive component `nand2`). These files are included in Appendix A.

The specified behavior for this circuit was derived using both a carry propagation technique, as discussed in the text by Hill and Peterson [16:575–585]. Given that the equations for the sum bits (S) and carry bits (C) are

$$\begin{aligned} S_j &= (((A_j \wedge B_j) \vee (\neg A_j \wedge \neg B_j)) \wedge C_{j-1}) \\ &\quad \vee (((A_j \wedge \neg B_j) \vee (\neg A_j \wedge B_j)) \wedge \neg C_{j-1}) \qquad (5.5) \\ C_j &= (A_j \wedge B_j) \vee (((A_j \wedge \neg B_j) \vee (\neg A_j \wedge B_j)) \wedge C_{j-1}) , \qquad (5.6) \end{aligned}$$

where S_0 and C_0 are the least-significant sum and carry bits generated and carry in is C_{-1}, we can simplify these by introducing a propagate function, P, and a generate function, G.

$$\begin{aligned} G_j &= A_j \wedge B_j \qquad (5.7) \\ P_j &= A_j \oplus B_j \\ &= (A_j \wedge \neg B_j) \vee (\neg A_j \wedge B_j) \qquad (5.8) \end{aligned}$$

The use of P and G allow us to rewrite Equations (5.5) and (5.6), *i.e.*,

$$C_j = G_j \vee (P_j \wedge C_{j-1}) \tag{5.9}$$

$$S_j = P_j \oplus C_{j-1} \tag{5.10}$$

$$= (\neg P_j \wedge C_{j-1}) \vee (P_j \wedge \neg C_{j-1}) \, . \tag{5.11}$$

Using Equations (5.9) and (5.11), we can derive an overall behavioral specification for a four-bit full adder [16:576–577]. First, a set of equations can be established for the carry bits:

$$C_0 = G_0 \vee (P_0 \wedge C_{in}) \tag{5.12}$$

$$C_1 = G_1 \vee (P_1 \wedge C_0)$$

$$= G_1 \vee (P_1 \wedge G_0) \vee (P_0 \wedge C_{in}) \tag{5.13}$$

$$C_2 = G_2 \vee (P_2 \wedge C_1)$$

$$= G_2 \vee (P_2 \wedge G_1) \vee (P_2 \wedge P_1 \wedge G_0) \vee (P_2 \wedge P_1 \wedge P_0 \wedge C_{in}) \tag{5.14}$$

$$C_{out} = G_3 \vee (P_3 \wedge C_2)$$

$$= G_3 \vee (P_3 \wedge G_2) \vee (P_3 \wedge P_2 \wedge P_1 \wedge G_1)$$

$$\vee (P_3 \wedge P_2 \wedge P_1 \wedge G_0) \vee (P_3 \wedge P_2 \wedge P_1 \wedge P_0 \wedge C_{in}) \, . \tag{5.15}$$

Using these results, we can then produce simplified equations for the sum bits, S_j:

$$S_0 = (\neg P_0 \wedge C_{in}) \vee (P_0 \wedge \neg C_{in}) \tag{5.16}$$

$$S_1 = (\neg P_1 \wedge C_0) \vee (P_1 \wedge \neg C_0) \tag{5.17}$$

$$S_2 = (\neg P_2 \wedge C_1) \vee (P_2 \wedge \neg C_1) \tag{5.18}$$

$$S_3 = (\neg P_3 \wedge C_2) \vee (P_3 \wedge \neg C_2) \, . \tag{5.19}$$

We can further simplify the equation for C_{out}, the only carry bit observed by the external environment, by first noting that a two bit full adder only generates a carry out (C_1) when the sum exceeds 4_{10}, as shown by Equation (5.20).

$$C_{j+1} = (A_{j+1} \wedge B_{j+1}) \vee ((A_{j+1} \vee B_{j+1}) \wedge ((C_{j-1} \wedge A_j) \vee (C_{j-1} \wedge B_j) \vee (A_j \wedge B_j)) \tag{5.20}$$

Application of this observation results in Equation (5.21) for C_1 in terms of C_{in}, eliminating the need to actually generate C_0 in order to determine C_1. We can then repeat this process to obtain Equation (5.23), obtaining the final carry out as a simple combinational logic function of the input signals [17:181]. By the same approach, after obtaining C_0 by means of Equations (5.6) or (5.9) we can derive a new representation for C_2, as indicated in Equation (5.25).

$$C_1 = (A_1 \wedge B_1) \vee ((A_1 \vee B_1) \wedge ((C_{in} \wedge A_0) \vee (C_{in} \wedge B_0) \vee (A_0 \wedge B_0))) \quad (5.21)$$

$$C_{out} = (A_3 \wedge B_3) \vee ((A_3 \vee B_3) \wedge ((C_1 \wedge A_2) \vee (C_1 \wedge B_2) \vee (A_2 \wedge B_2))) \quad (5.22)$$

$$\begin{aligned} = {} & (A_3 \wedge B_3) \vee ((A_3 \vee B_3) \\ & \wedge ((((A_1 \wedge B_1) \vee ((A_1 \vee B_1) \wedge ((C_{in} \wedge A_0) \vee (C_{in} \wedge B_0) \vee (A_0 \wedge B_0)))) \wedge A_2) \\ & \vee (((A_1 \wedge B_1) \vee ((A_1 \vee B_1) \wedge ((C_{in} \wedge A_0) \vee (C_{in} \wedge B_0) \vee (A_0 \wedge B_0)))) \wedge B_2) \\ & \vee (A_2 \wedge B_2))) \end{aligned} \quad (5.23)$$

$$C_2 = (A_2 \wedge B_2) \vee ((A_2 \vee B_2) \wedge ((C_0 \wedge A_1) \vee (C_0 \wedge B_1) \vee (A_1 \wedge B_1))) \quad (5.24)$$

$$\begin{aligned} = {} & (A_2 \wedge B_2) \vee ((A_2 \vee B_2) \\ & \wedge ((((A_0 \wedge B_0) \vee (C_{in} \wedge ((\neg A_0 \wedge B_0) \vee (A_0 \wedge \neg B_0)))) \wedge A_1) \\ & \vee (((A_0 \wedge B_0) \vee (C_{in} \wedge ((\neg A_0 \wedge B_0) \vee (A_0 \wedge \neg B_0)))) \wedge B_1) \\ & \vee (A_1 \wedge B_1))) \end{aligned} \quad (5.25)$$

Equation (5.23) is used as the output equation for the carry out in the four-bit full adder `fadd4_cl.pro`, as indicated in Table 5.8. Equations (5.21) and (5.25) are used in the generation of output equations for S_2 and S_3 from Equations (5.18) and (5.19), Equation (5.12) is used with Equation (5.17) to generate an output equation for S_1, and Equation (5.16) was used to generate an output equation for S_0. The output equation for S_2 is shown in Table 5.9.

```
output_eqn(fadd4_cl,carryout(FA4CL) :=
          or( and( in03(FA4CL),in13(FA4CL)),
              and( or( in03(FA4CL),in13(FA4CL)),
                   or( or( and( or( and( in01(FA4CL),in11(FA4CL)),
                                    and( or( in01(FA4CL),in11(FA4CL)),
                                         or( or( and( carryin(FA4CL),
                                                      in00(FA4CL)),
                                                 and( carryin(FA4CL),
                                                      in10(FA4CL))),
                                             and( in00(FA4CL),
                                                  in10(FA4CL))))),
                                in02(FA4CL)),
                            and( or( and( in01(FA4CL),in11(FA4CL)),
                                     and( or( in01(FA4CL),in11(FA4CL)),
                                          or( or( and( carryin(FA4CL),
                                                       in00(FA4CL)),
                                                  and( carryin(FA4CL),
                                                       in10(FA4CL))),
                                              and( in00(FA4CL),
                                                   in10(FA4CL))))),
                                in12(FA4CL)),
                           and( in02(FA4CL),in22(FA4CL))))))).
```

Table 5.8. Definition of Carry Out in Terms of Input Signals

```
output_eqn(fadd4_cl,sum2(FA4CL) :=
       or( and( neg( or( and(in02(FA4CL),neg( in12(FA4CL))),
                         and( neg( in02(FA4CL)),in12(FA4CL)) )),
                or( and( in01(FA4CL),in11(FA4CL)),
                    and( or( in01(FA4CL),in11(FA4CL)),
                         or( or( and( carryin(FA4CL),in00(FA4CL)),
                                 and( carryin(FA4CL),in10(FA4CL))),
                         and( in00(FA4CL),in10(FA4CL)))))),
           and( or( and(in02(FA4CL),neg( in12(FA4CL))),
                    and( neg( in02(FA4CL)),in12(FA4CL)) ),
                neg( or( and( in01(FA4CL),in11(FA4CL)),
                         and( or( in01(FA4CL),in11(FA4CL)),
                              or( or( and( carryin(FA4CL),in00(FA4CL)),
                                      and( carryin(FA4CL),in10(FA4CL))),
                              and( in00(FA4CL),in10(FA4CL))))) ))) ).
```

Table 5.9. Definition of Sum Bit S_2 in Terms of Input Signals

VI. Results and Recommendations

6.1 Results of System Enhancements

This thesis has demonstrated that the verification of digital logic systems through the use of a PROLOG-based environment such as **AFIT_VERIFY** is both practical and advantageous. Through the use of formal verification rather than simulation, a VLSI circuit designer can prove that his or her circuit designs precisely conform to the circuit specifications, rather than simulated the circuit design against a subset of those specifications.

At the start of this thesis, **AFIT_VERIFY** was sufficiently developed to perform simple demonstrations of formal circuit verification. The user was required to load all of the necessary program modules into the PROLOG interpreter, to manually load the necessary circuit description files, and then to manually initiate the verification process. As such, it was unreasonable to assume that anyone who was not a highly experienced PROLOG programmer would be able to use the **AFIT_VERIFY** system.

This thesis effort advanced the capabilities of the **AFIT_VERIFY** system, primarily in the area of improved user interface. The capabilities of Quintus PROLOG were used to full advantage in providing a menu-based interface. A centralized parts library was established, and procedures for inserting components into the library and extracting them into user directories were supplied.

In addition to the work spent on the user interface, a number of simple VLSI circuits were translated into **AFIT_VERIFY** descriptions. Each of these circuits was submitted to the verification process (sample verification runs are attached in Appendix B), demonstrating that the formal verification of real-world circuit designs is a realistic goal. In particular, the **AFIT_VERIFY** standard parts library now contains two-, three-, four-, and five-input NAND gates, two-input NOR gates, registers, integer incrementers, two-to-1 and four-to-1 multiplexors, and-or-invert gates, one bit half adders, a variety of one bit full adders, and a four bit full adder. These parts are hierarchically constructed and demonstrate the various features of the **AFIT_VERIFY** environment.

During the course of this research, a number of logic errors in the PROLOG source code for **AFIT_VERIFY** were discovered. Due to the high degree of interdependence between the various procedures, this task consumed a larger amount of time than previously anticipated. In-depth understanding of how essential (and intricate) PROLOG procedures,

such as `setof/3`, operate was not obtained until late in this thesis effort, hampering efforts to track some of the more elusive errors. In addition, time was spent on the unsuccessful integration of **AFIT_VERIFY** into the PROLOG-in-Scheme environment, as discussed briefly in Chapter 5.2.2. Additionally, although work was done on adding homomorphic proof strategies to the verification algorithm, this work was hampered by difficulties in expressing arbitrary state machines within the **AFIT_VERIFY** framework.

Despite these difficulties, however, substantial work was performed in the areas of improving the robustness and useability of the **AFIT_VERIFY** environment. As presented in this document, **AFIT_VERIFY** is a practical system for verification of digital logic systems. Its use is only limited by the ability of the circuit designer to specify the outputs of a module in terms of its input signals and by the execution speed of the host computer.

6.2 Recommendations for Future Work

The work on this thesis has revealed a number of areas in which **AFIT_VERIFY** should receive future enhancements.

1. Additional parts should be added to the standard parts library. AFIT students and faculty in the VLSI design and testing area should be consulted in the selection of parts for the library. In this fashion, **AFIT_VERIFY** will grow into a tool which will meet the needs of the AFIT population. One suggested part would be the four-by-four bit multiplier that has been implemented by Mr Gene Howell as part of a digital radio frequency memory (DRFM) [20]. This part is available in a structural VHDL description which has been successfully simulated in VHDL and is being fabricated in Gallium Arsenide. This part would be essential to verifying either larger subsystems of the DRFM (development of which is an ongoing track of thesis research) or to verifying a full arithmetic logic unit.
2. The verification subsystem should be modified in order to allow the use of PROLOG "demons" in describing the behavioral specification. The use of "demons" (helper procedures that can be invoked, as necessary, to perform useful tasks for higher level procedures) in the definition of output equations (`output_eqn/2`) would greatly simplify the hardware designer's task of specifying an overall behavior for a complex module. Deriving such specifications was extremely difficult for a number of the digital systems investigated during this effort. (Tables 5.8 and 5.9 demonstrate the complexity that can be achieved by even simple output equations.)

3. The **AFIT_VERIFY** system currently supports signals of type Boolean and Integer. Barrow's VERIFY system included support for a bit-addressable integer type, as well as interconnection between Boolean and Integer signals. The addition of these ;tures is recommended for compatibility between the VHDL language and **AFIT_VERIFY** descriptions.

4. The AFIT Department of Electrical and Computer Engineering's VLSInet includes Sun Microsystems SUN-4 workstations with Quintus PROLOG Release 2.4.2. Since much of the VLSI design work that occurs at AFIT takes place on the VLSInet computers, installation of the **AFIT_VERIFY** system on the VLSInet (in particular, on the workstation ares.afit.af.mil) would allow the students and faculty to use **AFIT_VERIFY** as one of their design tools.

5. Along with placing **AFIT_VERIFY** on an accessible VLSInet workstation, future thesis efforts should work on integrating **AFIT_VERIFY** with other VLSI design tools. One tool that might merit special consideration for integration into an overall design tool suite is Captain Joseph Eicher's PROLOG-based circuit specialization program [13].

6. An automated translator between VHDL structural descriptions and PROLOG-based **AFIT_VERIFY** module descriptions should be written. This task will need to be performed by someone who is familiar (and fluent) in both VHDL and PROLOG. A prototype VHDL-to-PROLOG parser has previously been written by CPT Michael Dukes. Access to a VHDL-to-**AFIT_VERIFY** translator would allow the verification of many of the circuits designed at AFIT and throughout the Air Force.

7. Work should continue on expanding the number and type of proof strategies available to the **AFIT_VERIFY** system.

Appendix A. *Program Listings*

The following files constitute the development code for **AFIT_VERIFY**. The files are as follows:

`qverify.pl` Main program file which contains menus and driver code.

`boole2.pl` Boolean expansion code, based upon work by CPT Dukes [12].

`derbeh.pl` Derive module behavior.

`derstate.pl` Derive module's next state.

`eqbeh.pl` Determine behavioral equivalences.

`eval.pl` Canonicalization and evaluation clauses.

`multdyn.pl` Allows module definition clauses to be loaded from a hierarchical set of files. (This file resides in the parts library area.)

`opentail.pl` Performs operations on open lists.

`qops.pl` Quintus PROLOG operator definitions and system-dependent procedures, including most file-related operations.

In addition, a number of files are located in the AFIT_VERIFY parts library. These files contain the module definitions that users have explicitly entered into the component library, as well as some index files used to retrieve the components. These files are as follows:

`parts.verified` Contains a set of facts listing all components that have been previously verified and stored in the parts library.

`counter.pl` Contains the definition of a simple integer counter, as described by Barrow [2:65].

`faddxor.pl` Contains the definition of a full adder built from NAND and XOR gates.

`modfiles.list` Stores an open list containing the names of all other component files except the file of primitive components.

`primitive.pl` Contains the definitions of the primitive components REG (register), INC (incrementer), MUX (two-input multiplexor), NAND2 (two-input NAND), NOR2 (two-input NOR).

`xor.pl` Contains the definition of an exclusive or (XOR) built from two-inut NAND gates.

`inv.pl` Contains the definition for a simple inverter circuit build from a single two-input NAND gate.

`aoi.pl` Contains the definition of an AND-OR-INVERT circuit, as per the example provided in the Zycad Reference Manual [32:Ch 10,73].

`halfadd.pl` Contains the definition of a half adder constructed from inverters and two-input NANDs.

`nand3.pl` Contains the definition of a three-input NAND gate.

`nand4.pl` Contains the definition of a four-input NAND gate.

`nand5.pl` Contains the definition of a five-input NAND gate.

`mux_4x1.pl` Contains the definition of a four-to-one multiplexor constructed from two-input multiplexor (MUX) primitives.

`halfadd.pl` Contains the definition of a half adder built from two-input NANDs and inverters.

`faddnor.pl` Contains the definition of a full adder constructed from inverters, half adders, and two-input NOR gates.

`fadd4_cl.pl` Contains the definition of a four bit full adder with carry lookahead built from half adders, inverters, exclusive ors, and two-, three-, four-, and five-input NAND gates.

A.1 Source Code Listings

A.1.1 qverify.pl

```
/*********************************************************/
/*                    QVERIFY.PL                         */
/*  This file consults the appropriate files to start the */
/*  AFIT_VERIFY environment in Quintus Prolog.  This is   */
/*  invoked by typing [qverify] at the Quintus prompt.    */
/*********************************************************/

:- asserta(library_directory('/usr/users/ela/labovitz/NewVerify/Work/Parts')).
                                        % Components live in the
                                        %  Parts subdirectory
                                        % -- This directory may be anywhere,
                                        %    but should be hardwired!

:- asserta(library_directory('.')).     % New Components should live in
                                        %  user's current directory

:- [qops,library(multdyn),eval].         % Consult the various files
:- [derbeh,derstate,boole2,eqbeh,opentail].

/*********************************************************** */
/*                                                          */
/*  Do_verify/0 provides the main program loop, running     */
/*  main_menu/0 to provide a user interface for the entire  */
/*  AFIT_VERIFY system.  Main_Menu/0 uses the helper        */
/*  procedure main_menu_choices to actually present the     */
/*  menu to the user.  When a user selects the 'verify'     */
/*  option from the menu system, the do_verification/1       */
/*  clause invokes other clauses to recursively verify each */
/*  Module.  Other clauses used include the following:      */
/*                                                          */
/*    derive_and_equate_behaviors:  provides mechanism to   */
/*        derive behavior for each output, and determine    */
/*        equivalence to specified behavior.                */
/*    derive_and_equate_states:  provides mechanism to      */
/*        derive behavior for each next state, and          */
/*        determine equivalence to specified next state.    */
/*    verify_components:  uses verify to recursively        */
/*        verify components prior to deriving component     */
/*        behavior and next state.                          */
/*                                                          */
/*  These clauses use the fail, always true combination     */
/*  to succeed for all possible outputs and next states.    */
/*********************************************************/
```

```
/*   main prompting loop.    */

do_verify :-
    nl,nl,
    writeln['             Performing AFIT_VERIFY Verification!']),
    nl,
    main_menu,
    do_verify.

/************************************************************/

main_menu :-
    main_menu_choices(Answer),                % display menu & get choice
    ((Answer==preload) ->           %% CHOICE=PRELOAD
      ((not flag(parts_loaded)) ->            % if verified parts have not been
                                              % previously loaded, then ask if
                                              % we should load them now.  If
                                              % yes, invoke load_known_parts.
        ( yesno('Should I preload the previously verified components? y/n ',
                 y),
          load_known_parts
          |
        nl)                                   % not loaded, but don't want to
        |
        writeln(['Already preloaded....']),   % If no, satisfy the partial goal
                                              % and continue onward.
    nl)
    |
    (Answer==reverify) ->           %% CHOICE=REVERIFY
      (get_verified_parts(PartsOpenList),
       convert_to_notail(PartsOpenList,PartsList),
       closed_flatlist_to_string(PartsList,PartsString),
       string_append('Choices: ',PartsString,Prompt),
       ask_oneof(Prompt,PartsList,Component),
        do_verification(Component))           % Re-verify the part
    |
    (Answer==halt) ->               %% CHOICE=HALT
      conditional_halt
    |
    (Answer==list) ->               %% CHOICE=LIST
      list_known_parts
    |
    (Answer==verify) ->             %% CHOICE=VERIFY
```

```
      ( ask_for_term(
         'Name of module (file) to be verified (do not include .pl suffix): ',
         [' Enter a module (file) name at the prompt.  However,',nl,
          ' do _not_ enter the file suffix (usually .pl).',nl,
          ' Enter the keyword ''exit'' to quit back to the menu.'],
         ModuleName),
(ModuleName \== exit) ->
           (do_verification(ModuleName)) % read in new file & verify
        |
           true)
    |
    (Answer==insert) ->            %% CHOICE=INSERT
      ( ask_for_term(
   'Module name to be inserted in library (do not include .pl suffix): ',
         [' Enter a module (file) name at the prompt.  However,',nl,
          ' do _not_ enter the file suffix (usually .pl).',nl,
          ' Enter the keyword ''exit'' to quit back to the menu.'],
         ModuleName),
(ModuleName \== exit) ->
           copy_new_module(ModuleName)
        |
         true )
    |
    (Answer==extract) ->           %% CHOICE=EXTRACT
      ( ask_for_term(
   'Module name to be extracted from library (do not include .pl suffix): ',
         [' Enter a module (file) name at the prompt.  However,',nl,
          ' do _not_ enter the file suffix (usually .pl).',nl,
          ' Enter the keyword ''exit'' to quit back to the menu.'],
         ModuleName),
(ModuleName \== exit) ->
           extract_old_module(ModuleName)
        |
         true )
   ).

/*************************************************************/

main_menu_choices(MenuChoice) :-
    writeln(
      ['Select your action from the following choices:']),
    writeln(
      ['   Preload the previously verified components into the database']),
    writeln(
      ['         (This may increase execution speed of a verification run)']),
```

```
    writeln(
      ['   Reverify a component from the component library']),
    writeln(
      ['   List the nonprimitive components which have been verified ',
       'this session']),
    writeln(
      ['   Insert a component into the component library area']),
    writeln(
      ['   Extract a component from the library area into current directory']),
    writeln(
      ['   Verify a new component from the current directory']),
    writeln(
      ['   Halt the program and exit Prolog']),
    writeln(
      ['         (Note: this option _is_ revocable at the next menu!)']),
    nl,
    ask_oneof(
  'Enter your choice: preload, reverify, list, verify, insert, extract, halt',
              [preload,reverify,list,insert,verify,extract,halt],
              MenuChoice).

/*************************************************************/
/* Do_Verification(ComponentFile) is the control procedure for the      */
/*   verification process.  After establishing whether the verification */
/*   run is to be in verbose or terse mode, the various component and   */
/*   subcomponent files are loaded into working memory.  If the top     */
/*   level component is not known to be previously verified, the        */
/*   verification process starts.  If it is successfully verified, the  */
/*   user is asked whether or not the top component should be inserted  */
/*   into the parts library.                                            */


do_verification(ComponentFile) :-
    retractall(flag(terse)),          % reset terse-mode flag
    ask_oneof('Should this verification run be executed in TERSE mode?',
[yes,no],yes,TerseFlag),
    ((TerseFlag==yes) ->
asserta(flag(terse))
     |
     true),
    reconsult(library(multdyn)),      % reset multifile/1 and dynamic/1
    get_top(ComponentFile),           % load the ComponentFile into memory
    writeln(['Component file ',ComponentFile,' loaded....']),
    module_name(Component),           % get the name of the top component
    writeln(['--- Beginning verification of module ',Component]),
```

```
    nl,
    (not flag(verified(Component)) ->    % if not previously verified, then...
      (!,
       verify(Component),
       writeln(['>>>> Component ',Component,' verified! <<<<',nl]),
       get_verified_parts(PartsOpenList),
       convert_to_notail(PartsOpenList,PartsList),
       (not member(Component,PartsList) ->
          (ask_oneof('Should this component be inserted into the library? ',
                     [yes,no],no,Answer),
           (Answer==yes  ->
              copy_new_module(ComponentFile)
           |
              true ))
        |
          true))
    |
     writeln([nl,'>>>> Component ',Component,' already verified! <<<<',nl])).

do_verification(_ComponentFile) :-
    module_name(Component),                % We only reach this clause if
    nl,                                    %  the component can't be verified
    writeln(['****** Component ',Component,' fails verification! ******']),
    nl,
    nl.

/*************************************************************/

conditional_halt :-
    yesno('Do you really want to halt Prolog? y/n ',n),
    halt.

conditional_halt :-
    do_verify.                    % Previous clause failed, so repeat loop

/*************************************************************/

/* ask_for_term(Prompt,Help,Term) prompts the user for the input  */
/*   Term, providing the initial Prompt and any additional Help   */
/*   whenever the input ? is given by the user.                   */

ask_for_term(PromptConstant,HelpList,Term) :-
    prompted_constant(PromptConstant,TempTerm),
    ( TempTerm = '.' ->
      ( writeln(['  NULL INPUT.  PLEASE REENTER.']),
```

```
        nl,
        ask_for_term(PromptConstant,HelpList,Term))
    |
    ( TempTerm = '?' ->
      ( writeln(HelpList),
        nl,
        ask_for_term(PromptConstant,HelpList,Term)))
    |
      Term=TempTerm).


/*************************************************************/
/* Verify/1 is the interface from the initial menu system    */
/* into the ''real'' verification subsystem.  Multiple       */
/* clauses are provided to cover all possible cases of       */
/* primitive/nonprimitive state/stateless modules.           */
/*************************************************************/

verify(Module) :-
   flag(verified(Module)),           %  This is a previously verified module
   !,
   writeln(['>>>',Module,' previously verified >>>']),
   nl.

/*************************************************************/
/*  For a primitive module, behavior = structure.            */
/*  No need to reassert this in the database, since it can   */
/*  be taken from the behavioral specification.              */
/*  output_eqn(Module, Output := Behavior)  ALREADY EXISTS.  */
/*  If it is decided that derived_behavior should appear as  */
/*  asserta(derived_behavior(Module,Output,Behavior)), the   */
/*  derive_behavior clause dealing with primitives can be    */
/*  removed.  A similar decision is required for the         */
/*  asserta(flag(verified(Module))) for a primitive Module.  */
/*  With only one possible verified clause per Module,       */
/*  the space required was minimal for the time savings.     */
/*************************************************************/

verify(Module) :-
   not part(Module,_,_),             % primitive module with no state
   not state_eqn(Module,_),
   !,
   asserta(flag(verified(Module))),
   writeln(['>>>',Module,' primitive (needs no verification)>>>']),
   nl.
```

```
/*************************************************************/
/*  For a primitive module, behavior = structure, as above. */
/*  Also, no need to reasert next state either.              */
/*  state_eqn(Module,Nextstate := Function) ALREADY EXISTS   */
/*************************************************************/

verify(Module) :-
   not part(Module,_,_),              % primitive module with state
   !,
   asserta(flag(verified(Module))),
   writeln(['>>>',Module,' primitive (needs no verification)>>>']),
   nl.

/*************************************************************/
/* Derive behavior for all outputs and if equal to           */
/* specified behavior, then assert in database.  This may    */
/* require later garbage collection if the earlier outputs   */
/* are okay, but a later output is not.  This would require  */
/* a cleanup to check the flag(verified(Module)) against the */
/* derived_behavior and next_state clauses. If the clauses   */
/* are not consistent, then remove all derive_behavior and   */
/* next_state clauses.                                       */
/*************************************************************/

verify(Module) :-
   not state_eqn(Module,_),           % non-primitive module with no state
   nl,
   writeln(['>>> Attempting to verify non-primitive module ',Module,'>>>']),
   verify_components(Module),
   !,
   derive_and_equate_behaviors(Module),
   asserta(flag(verified(Module))),
   writeln(
       ['<<< Success! Behavior of ',Module, ' meets its specification.<<<']),
   nl.


verify(Module) :-                  % This must be a non-primitive with state
   nl,
   writeln(['>>> Attempting to verify non-primitive module ',Module,'>>>']),
   nl,
   verify_components(Module),
   !,
   derive_and_equate_behaviors(Module),
```

```
   derive_and_equate_states(Module),
   asserta(flag(verified(Module))),
   writeln(
       ['<<< Success! Behavior of ',Module, ' meets its specification.<<<']),
   nl.


/*************************************************************/
/*  derive_and_equate_behaviors                              */
/*************************************************************/


/*************************************************************/
/*  The first clause of the next three procedures always     */
/*  succeeds.  We need a way to check that all components    */
/*  are verified, and all behaviors and next_states are      */
/*  equivalent.  The second clause of each procedure does    */
/*  this checking by generating lists of components, states, */
/*  and outputs and comparing the length of the two lists    */
/*  for the states and outputs since no two state or output  */
/*  names should be generated twice for a single module.     */
/*  Setof will generate all Components for a single module,  */
/*  so we just check to see if that component was actually   */
/*  verified.                                                */
/*************************************************************/

derive_and_equate_behaviors(Module) :-
   derive_behaviors(Module,Output,Derived_Beh),
   equal_behaviors(Module,Output,Derived_Beh),
   asserta(derived_behavior(Module,Output,Derived_Beh)),
   fail.

derive_and_equate_behaviors(Module) :-   % added existentials to  setof
   setof(Outputs,Dummy1^output_eqn(Module,Outputs := Dummy1),Outlist),
   length(Outlist,Outnum),
   setof(Outputs,Dummy2^derived_behavior(Module,Outputs,Dummy2),Derlist0),
   setof_to_trueset(Derlist0,Derlist),   % make Derlist into a true set
   length(Derlist,Dernum),
   !,
   writeln([nl,'For module ',Module,' :']),
   writeln([' Specified output list is ', Outlist]),
   writeln([' Derived output list is   ', Derlist]),
   writeln(['  Number of specified outputs is ', Outnum]),
   writeln(['  Number of derived ouputs is    ', Dernum,nl]),
   Outnum =:= Dernum,              % same number of outputs
```

```
   unifiable_lists(Outlist,Derlist),      % same output functors
   writeln(['   ',Outlist,' matches with ',Derlist,nl]).

derive_and_equate_behaviors(Module) :-
   retract(derived_behavior(Module,_,_)),
   fail.

/*************************************************************/
/* setof_to_trueset(AlmostSet,TrueSet) is used to make a     */
/*   'set' AlmostSet with uninstantiated variables returned  */
/*   by setof into a TrueSet where only one instance of      */
/*   any element (without regard to uninstantiated           */
/*   variables) is found.                                    */
/*************************************************************/

setof_to_trueset([],[]) :- !.

setof_to_trueset([Head|Tail],TSet) :-
   unifiable_with_list(Head,Tail),  % is another instance of Head in Tail
   !,
   setof_to_trueset(Tail,TSet).

setof_to_trueset([Head|Tail],[Head|TSet]) :-
   setof_to_trueset(Tail,TSet).     % no other instance of Head in Tail


/*************************************************************/
/*  derive_and_equate_states                                 */
/*************************************************************/

derive_and_equate_states(Module) :-
   derive_states(Module,State,Next_State),
   equal_states(Module,State,Next_State),
   asserta(next_state(Module,State,Next_State)),
   fail.

derive_and_equate_states(Module) :-
   setof(States,state_eqn(Module,States := _),Statelist),
                                  % get set of all specified States
   length(Statelist,Statenum),
   setof(DStates,next_state(Module,DStates,_),Derlist),
                                  % get set of all derived States
   length(Derlist,Dernum),
   !,
   writeln([nl,'For module ',Module,' :']),
```

```
   writeln([' Specified State list is ', Statelist]),
   writeln([' Derived State list is   ', Derlist]),
   writeln(['  Number of Specified States is ', Statenum]),
   writeln(['  Number of Derived States is   ', Dernum]),
   nl,
   Statenum =:= Dernum,                    % same number of states
   unifiable_lists(Statelist,Derlist),     % same output functors
   writeln(['   ',Statelist,' matches with ',Derlist,nl]).


derive_and_equate_states(Module) :-
   retract(next_state(Module,_,_)),
   fail.

/*************************************************************/
/* unifiable_lists/2 checks if the two list arguments can    */
/*   be unified.  It uses unifiable_with_list/2 to check     */
/*   each element of the first list, in turn, with the       */
/*   list.  We assume that we have already checked that the  */
/*   two lists have the same length, and that they are all   */
/*   unique elements.                                        */
/*************************************************************/

unifiable_lists([],_List2) :- !.

unifiable_lists([Head|Tail],List2) :-
   unifiable_with_list(Head,List2),
   unifiable_lists(Tail,List2).

unifiable_with_list(Element,[Head|_Tail]) :-
   not(not(Element = Head)),           % can we unify Element and Head?
   !.

unifiable_with_list(Element,[_Head|Tail]) :-
   unifiable_with_list(Element,Tail).   % otherwise, check again.

/*************************************************************/
/* verify_components/1 verifies each of the subcomponents    */
/*  of the current module, in turn.  When completed, it      */
/*  uses parts_verified/2 to check if all of the             */
/*  subcomponents were, in fact, successfully verified.      */
/*************************************************************/

verify_components(Module) :-
   part(Module,_,Component), % get a subcomponent,
```

```
   verify(Component), %  verify it, and try
   fail. %  to get another subcomponent

verify_components(Module) :-
   setof(Component,Name^part(Module,Name,Component),Complist),
% Complist is list of all
%  submodules
   parts_verified(Complist,Complist), % Are all submodules verified?
   !,
   writeln(['+> Module ',Module,' has verified submodules: ',
            Complist]),
   nl.

verify_components(Module) :-
   setof(Component2,Name^part(Module,Name,Component2),CompList),
% Complist is list of all
%  submodules
   parts_verified(CompList,VCompList),
   writeln(['-> Module ',Module,' can only verify submodules: ',
    VCompList,nl,'   out of submodules: ',CompList, ' <-']),
   nl.

/*************************************************************/
/*                                                           */
/*  parts_verfied - This procedure ensures that all parts    */
/*   (Components) of a Module have been verified.  A list    */
/*   of parts(Components) generated by setof is passed,      */
/*   and parts_verifed checks that each one has an asserted  */
/*   verified fact and returns a list of those which have,   */
/*   in fact, been matched with such a fact.                 */
/*                                                           */
/*************************************************************/

parts_verified([],[]) :- !.     % base case for recursion

parts_verified([Component|Tail],[Component|Rest]) :-
   flag(verified(Component)),     % is Component verified:
   !,
   parts_verified(Tail,Rest).     % if so, check tail...

parts_verified([_Component|Tail],Rest) :- % only reach this case if
   parts_verified(Tail,Rest).     % can't verify a Component

/*************************************************************/
/*************************************************************/
```

```
/*************************************************************/

/* Restart/0 writes an initial welcome message and starts   */
/*    the main program loop.                                 */

restart :-
    writeln(['                  Welcome to AFIT_VERIFY!']),
    writeln(['                  ======================',nl]),
    writeln(
   ['          (Type ? at any prompt if you require help)',nl]),
    do_verify.

/* The following directive will start the program when this */
/* file is consulted.                                       */

%:- restart.

/* Using the following directive INSTEAD of the previous    */
/* directive will save the compiled program into an         */
/* executable named AFIT_Verify.  Running AFIT_Verify will  */
/* then run the restart/0 procedure and start the program.  */

:- save('AFIT_Verify',1),  % Save compiled program
    restart.

/*************************************************************/
```

A.1.2 boole2.pl

```
/*************************************************************/
/*                      BOOLE2.PL                            */
/*                                                           */
/*                  BOOLE'S EXPANSION                        */
/*                                                           */
/*  The following code is a modified version of code         */
/*  developed by CPT Mike Dukes.                             */
/*                                                           */
/*  He defined xor($), or(@), and(^), and not(~) as          */
/*  operators.  Sparks used them as principle functors.      */
/*  A detailed explanation of this code can be found in      */
/*  CPT Dukes PROVING BOOLEAN EQUIVALENCE WITH PROLOG.       */
/*  Additional in-line comments are provided where           */
/*  modification for this approach is required.              */
/*                                                           */
/*  Currently, the functors xor/2, or/2, and/2, neg/1,       */
/*  nor/2 and nand/2 are supported.                          */
/*************************************************************/

eval(or(1,_),1):-!.
eval(or(_,1),1):-!.
eval(or(0,X),X):-!.
eval(or(X,0),X):-!.
eval(or(neg(X),X),1):-!.
eval(or(X,neg(X)),1):-!.
eval(or(X,X),X):-!.

eval(and(1,X),X):-!.
eval(and(X,1),X):-!.
eval(and(0,_),0):-!.
eval(and(_,0),0):-!.
eval(and(neg(X),X),0):-!.
eval(and(X,neg(X)),0):-!.
eval(and(X,X),X):-!.

eval(xor(X,1),neg(X)):-!.
eval(xor(1,X),neg(X)):-!.
eval(xor(0,X),X):-!.
eval(xor(X,0),X):-!.
eval(xor(X,X),0):-!.
eval(xor(neg(X),X),1):-!.
eval(xor(X,neg(X)),1):-!.

eval(neg(0),1):-!.
```

```
eval(neg(1),0):-!.

eval(X,X):-!.

/* Added support for 2-input nand and nor as evaluatable functors. */

eval(nor(1,_),0):-!.
eval(nor(_,1),0):-!.
eval(nor(0,X),neg(X)):-!.
eval(nor(X,0),neg(X)):-!.
eval(nor(neg(X),X),0):-!.
eval(nor(X,neg(X)),0):-!.
eval(nor(X,X),neg(X)):-!.

eval(nand(1,X),neg(X)):-!.
eval(nand(X,1),neg(X)):-!.
eval(nand(0,_),1):-!.
eval(nand(_,0),1):-!.
eval(nand(neg(X),X),1):-!.
eval(nand(X,neg(X)),1):-!.
eval(nand(X,X),neg(X)):-!.

/* extract(Var,Expression) returns a variable Var selected from   */
/*     those found in the structure Expression, using a blind,    */
/*     depth-first search.                                        */

extract(X,X) :-          % X is of the form inX(Avariable)
  X =.. [_,Arg],
  var(Arg),!.

extract(X,neg(Y)) :-
  extract(X,Y).

extract(X,or(L,_)) :-
  extract(X,L).

extract(X,or(_,R)) :-
  extract(X,R).

extract(X,and(L,_)) :-
  extract(X,L).
extract(X,and(_,R)) :-
  extract(X,R).

extract(X,xor(L,_)) :-
```

```
  extract(X,L).
extract(X,xor(_,R)) :-
  extract(X,R).

extract(X,nor(L,_)) :-
    extract(X,L).
extract(X,nor(_,R)) :-
    extract(X,R).

extract(X,nand(L,_)) :-
    extract(X,L).
extract(X,nand(_,R)) :-
    extract(X,R).

/* remove_x_1(OldExp,Var,NewExp) and remove_x_0(OldExp,Var,NewExp)   */
/*      replace every occurrance of the variable Var in structure    */
/*      OldExp with 1 (or 0) in the structure NewExp.                */

remove_x_1(Y,_X,Y) :-
  atomic(Y),
  !.

remove_x_1(neg(Y),_X,neg(Y)) :-
  atomic(Y),
  !.

remove_x_1(Y,X,Y) :-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I2,_Arg2],                    % we already know X is a Variable
  I1 \== I2,!.

remove_x_1(neg(Y),X,neg(Y)) :-
  Y =.. [I1,_Arg],
  var(_Arg1),
  X =.. [I2,_Arg2],                    % we already know X is a Variable
  I1 \== I2,!.

remove_x_1(Y,X,1):-
  Y =.. [I1,Arg1],                     % in0(_1) \== in0(_2) in Prolog
  var(Arg1),                           %   but we know it is the same input
  X =.. [I1,_Arg2],!.                  % we already know X is a Variable

remove_x_1(neg(Y),X,0):-
  Y =.. [I1,Arg1],
```

```
  var(Arg1),
  X =.. [I1,_Arg2],!.                    % we already know X is a Variable

remove_x_1(neg(Y),X,neg(NewY)) :-
  !,
  remove_x_1(Y,X,NewY).

remove_x_1(or(L,R),X,or(LNew,RNew)) :-
  !,
  remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).

remove_x_1(and(L,R),X,and(LNew,RNew)) :-
  !,
  remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).

remove_x_1(xor(L,R),X,xor(LNew,RNew)) :-
  !,
  remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).

remove_x_1(nor(L,R),X,nor(LNew,RNew)) :-
  !,
  remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).

remove_x_1(nand(L,R),X,nand(LNew,RNew)) :-
  !,
  remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).

remove_x_0(Y,_X,Y) :-
  atomic(Y),
  !.

remove_x_0(neg(Y),_X,neg(Y)) :-
  atomic(Y),
  !.

remove_x_0(Y,X,Y) :-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I2,_Arg2],                      % we already know X is a Variable
  I1 \== I2,!.
```

```
remove_x_0(neg(Y),X,neg(Y)) :-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I2,_Arg2],                    % we already know X is a Variable
  I1 \== I2,!.

remove_x_0(Y,X,0):-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I1,_Arg2],!.                  % we already know X is a Variable

remove_x_0(neg(Y),X,1):-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I1,_Arg2],!.                  % we already know X is a Variable

remove_x_0(neg(Y),X,neg(NewY)) :-
  !,
  remove_x_0(Y,X,NewY).

remove_x_0(or(L,R),X,or(LNew,RNew)) :-
  !,
  remove_x_0(L,X,LNew),
  remove_x_0(R,X,RNew).

remove_x_0(and(L,R),X,and(LNew,RNew)) :-
  !,
  remove_x_0(L,X,LNew),
  remove_x_0(R,X,RNew).

remove_x_0(xor(L,R),X,xor(LNew,RNew)) :-
  !,
  remove_x_0(L,X,LNew),
  remove_x_0(R,X,RNew).

remove_x_0(nor(L,R),X,nor(LNew,RNew)) :-
  !,
  remove_x_0(L,X,LNew),
  remove_x_0(R,X,RNew).

remove_x_0(nand(L,R),X,nand(LNew,RNew)) :-
  !,
  remove_x_0(L,X,LNew),
  remove_x_0(R,X,RNew).
```

```
/*  divide(F,X,F0,F1) is used to expand the clause F into two clauses, */
/*    F0Temp (in which every X is replaced by logical 0) and F1Temp    */
/*    (in which every X is replaced by logical 1), which are then      */
/*    simplified by evaluate_dukes/2.                                  */

divide(F,X,F0,F1) :-
  remove_x_0(F,X,F0Temp),
  remove_x_1(F,X,F1Temp),
  evaluate_dukes(F0Temp,F0),
  evaluate_dukes(F1Temp,F1).

/*  evaluate_dukes(Expr,NewExpr) performs an elementary simplification */
/*     of structure Expr into NewExpr, using itself recursively and    */
/*     eval/2.                                                         */


evaluate_dukes(X,X) :-
  atomic(X),!.

evaluate_dukes(X,X):-
  X =.. [I,Arg],
  var(Arg),
  I \== neg,
  !.

evaluate_dukes(neg(F),FReduced) :-
  evaluate_dukes(F,FTemp),
  eval(neg(FTemp),FReduced),!.

evaluate_dukes(or(L,R),Resolved) :-
  evaluate_dukes(L,LNew),
  evaluate_dukes(R,RNew),
  eval(or(LNew,RNew),Resolved).

evaluate_dukes(and(L,R),Resolved) :-
  evaluate_dukes(L,LNew),
  evaluate_dukes(R,RNew),
  eval(and(LNew,RNew),Resolved).

evaluate_dukes(xor(L,R),Resolved) :-
  evaluate_dukes(L,LNew),
  evaluate_dukes(R,RNew),
  eval(xor(LNew,RNew),Resolved).
```

```
evaluate_dukes(nor(L,R),Resolved) :-
    evaluate_dukes(L,LNew),
    evaluate_dukes(R,RNew),
    eval(nor(LNew,RNew),Resolved).

evaluate_dukes(nand(L,R),Resolved) :-
    evaluate_dukes(L,LNew),
    evaluate_dukes(R,RNew),
    eval(nand(LNew,RNew),Resolved).

/*   eq(Expression1,Expression2) tests if Expression1 and Expression2 are */
/*      unifiable after Boolean Expansion (performed by extract/2 and     */
/*      divide/4).                                                        */

eq(X,X):-!.
eq(F,G) :-
  extract(X,F),
  divide(F,X,F0,F1),
  divide(G,X,G0,G1),!,
  eq(F0,G0),!,
  eq(F1,G1),!.
```

A.1.3 derbeh.pl

```
/*************************************************************/
/*            DERBEH.PL    Derive_Behaviors                  */
/*                                                           */
/*    The two derive_behaviors clauses identify a specific   */
/*  output for the Module.  The derive_behavior clauses      */
/*  are then invoked to derive the behavior of that output   */
/*  for this particular Module.                              */
/*    The derive_behavior clause uses the part, connected,   */
/*  and output_eqn clauses to derive an output's behavior    */
/*  and tie it to this instantiation of the module as        */
/*  described in the in-line comments to follow.             */
/*    The derive_and_equate_behaviors clause in verify.pl    */
/*  uses derive_behaviors to derive all Module outputs and   */
/*  determine their equivalence to the specified output.     */
/*    The arguments for derive_behaviors and derive_behavior*/
/*  have the following meanings:                             */
/*                                                           */
/*  Args:  Module:  e.g., 'xor','nand2', ...                 */
/*         Form:  A formula involving terminal-behavior.     */
/*                   In the initial query, this may be       */
/*                   something like 'out(X)'.                */
/*         Behavior: The resulting derived behavior.         */
/*  In the present version of this procedure, it is assumed  */
/*  that all of the component-parts of Module have been      */
/*  previously verified by verify_components.  The verified  */
/*  components derived behavior is either asserted in a      */
/*  derived_behavior clause or specified in an output_eqn    */
/*  if the component-part is a primitive.                    */
/*                                                           */
/*************************************************************/

/*************************************************************/
/* derive_behaviors(Module,Form,Behavior)                    */
/*   Case I: no state equation provided for module           */
/*************************************************************/

derive_behaviors(Module,Form,Behavior) :-
   not state_eqn(Module,_),
   !,
   output_eqn(Module,Form := _Spec_Behavior),
   derive_behavior(Module,Form,Behavior).

/*************************************************************/
/*  Case II: state equation is provided for module           */
```

```
/*************************************************************/

derive_behaviors(Module,Form,Behavior) :-
   output_eqn(Module,Form := _Spec_Behavior),
   derive_behavior(Module,Form,TBehavior),
   substitute_state(Module,TBehavior,Behavior).

/*    Note:  This might require the removal of some        */
/*           internal variables at a later time.           */
/*************************************************************/


/*************************************************************/
/* derive_behavior(Module,Form,Source) :-                    */
/* Rules 1A and 1B derive behavior if Form is the name of a  */
/* terminal to which some other terminal, in Module, is      */
/* connected.  Rule 1A is invoked if the other terminal is a */
/* primary terminal in Module, i.e., one of its inputs or    */
/* outputs. Rule 1B is invoked if the other terminal belongs */
/* to one of Module's component-parts.                       */
/*************************************************************/

derive_behavior(Module, Form, Source) :-
   connected(Module, Source, Form),
   primary_source(Source),
   !,
   (not flag(terse) ->
        writeln(['Applying Derive_Behavior Rule 1A to ',Form,nl])
    |
    true).

derive_behavior(Module, Form, Behavior) :-
   connected(Module, Source, Form),
   derived_source(Source),
   !,
   (not flag(terse) ->
        writeln(['Applying Derive_Behavior Rule 1B to ', Form])
    |
    true),
   derive_behavior(Module, Source, Behavior).

/*************************************************************/
/* Rule 2 is invoked if Form is the name of a terminal of one */
/* of Module's component-parts.   Rule 2A handles primitive   */
/* components where Rule 2B handles non-primitive components. */
```

```
/* The only real difference is where to locate the Components */
/* derived behavior  (output_eqn vs derived_behavior).  If it */
/* is later decided to assert a derived_behavior clause for   */
/*  primitives, then Rule 2A can be removed.                  */
/************************************************************/

derive_behavior(Module, Form, Behavior) :-
   Form \== 1,
   Form =.. [_F,G],
   part(Module, G, Component),
   not part(Component,_,_),                % Component is a primitive module
   output_eqn(Component, Form := OutForm),
   !,
   writeln(['Applying Derive_Behavior Rule 2A to ', Form,
            ' of',nl,'   primitive component ',Component,':']),
   writeln([' ',Component,'''s output equation:']),
   writeln(['   ', Form, ' := ', OutForm,nl]),
   derive_behavior(Module, OutForm, Behavior).
/*   We have replaced the gate inputs with module variables  */


derive_behavior(Module, Form, Behavior) :-
   Form \== 1,
   Form =.. [_F,G],
   part(Module, G, Component),     % Since we passed the cut, this
                                   % is not a primitive component
                                   % that was previously verified
                                   % due to verify_components in
                                   % verify clause
   derived_behavior(Component,Form,OutForm),
   !,
   writeln(['Applying Derive_Behavior Rule 2B to ', Form,
            ' of',nl,'   nonprimitive component ',Component,':']),
   writeln([' ',Component,'''s derived behavior:']),
   writeln(['   ', Form, ' := ', OutForm]),
   derive_behavior(Module,OutForm,Behavior).
/*   We have replaced the gate inputs with module variables  */
/************************************************************/

/************************************************************/
/* The remaining rules cover cases in which FORM is not the  */
/* name of a terminal, but is a formula involving such name. */
/* This is where additional types of boolean or non-boolean  */
/* behavioral rules can be added in future work.  These rules*/
/* simplify internal components of a specified behavioral    */
```

```
/* structure.  NOTE: The evaluate1 clause is a simple        */
/* canonicalizer which should also be modified if new        */
/* behaviors are added.                                      */
/*************************************************************/

derive_behavior(Module, neg(Form), Behavior) :-
   !,
   (not flag(terse) ->
        writeln(['Applying Derive_Behavior Rule 3 to ',neg(Form)])
    |
    true),
   derive_behavior(Module, Form, Beh1),
   evaluate1(neg(Beh1), Behavior).

derive_behavior(Module, and(Form1,Form2), Beh) :-
   !,
   (not flag(terse) ->
        writeln(['Applying Derive_Behavior Rule 4 to ',and(Form1,Form2)])
    |
    true),
   derive_behavior(Module, Form1, Beh1),
   derive_behavior(Module, Form2, Beh2),
   evaluate1(and(Beh1,Beh2), Beh).

derive_behavior(Module, or(Form1,Form2), Beh) :-
   !,
   (not flag(terse) ->
        writeln(['Applying Derive_Behavior Rule 5 to ',or(Form1,Form2)])
    |
    true),
   derive_behavior(Module, Form1, Beh1),
   derive_behavior(Module, Form2, Beh2),
   evaluate1(or(Beh1,Beh2), Beh).

derive_behavior(Module, if(Cond,Texp,Fexp), Beh) :-
   !,
   (not flag(terse) ->
        writeln(['Applying Derive_Behavior Rule 6 to ',if(Cond,Texp,Fexp)])
    |
    true),
   derive_behavior(Module, Cond, NCond),
   derive_behavior(Module, Texp, NTexp),
   derive_behavior(Module, Fexp, NFexp),
   evaluate1(if(NCond,NTexp,NFexp), Beh).
```

```
/*********************************************************/
/*  SPARKS NOTE:: CHECK THIS                             */
/* We get an infinite loop when we try to derive the  */
/* behavior of counter and in(inca(Acounter)) because */
/* in(incA(Acounter)) in(Acounter) fails occurs check */
/*********************************************************/

 derive_behavior(Module, First + Second, Beh) :-
   !,
   (not flag(terse) ->
        writeln(['Applying Derive_Behavior Rule 7 to ',First + Second])
    |
    true),
   derive_behavior(Module, First, Beh1),
   derive_behavior(Module, Second, Beh2),
   evaluate1(Beh1 + Beh2, Beh).

/*********************************************************/
/* The default rules catch behavior which we haven't yet    */
/* described in a rule or which shouldn't be described.     */
/*********************************************************/

derive_behavior(_Module, Form, Form) :-
   (not flag(terse) ->
        writeln(['Applying default Derive_Behavior Rule to ',Form])
    |
    true).

/*********************************************************/
/*  primary_source and derived source distinguish between a  */
/*  Module input(primary_source) and a Component input       */
/*  (secondary_source).                                      */
/*********************************************************/

primary_source(Source) :-
   Source =.. [_,Arg],
   var(Arg).

derived_source(Source) :-
   Source =.. [_,Arg],
   Arg =.. [_,Arg2],
   var(Arg2).
```

A.1.4 derstate.pl

```
/*********************************************************/
/*            DERSTATE.PL   Derive_States                */
/*                                                       */
/*  The clause derive_states finds a state variable      */
/*  for a Module, how this state variable fits into the  */
/*  internal structure of the Module, derives the behavior*/
/*  of the internal structure, and substitutes the       */
/*  state variable name for the internal name whenever it */
/*  appears in the derived behavior.  The state_of,      */
/*  state_map, and state_eqn facts from the specified    */
/*  Module description are used to identify the          */
/*  appropriate variables. Then the two clauses          */
/*  derive_behavior and substitute_state are invoked to  */
/*  create the desired Next_State.                       */
/*                                                       */
/*********************************************************/

derive_states(Module,State,Next_State) :-
   state_of(Module,State,_Type),              % this has state information
   state_map(Module,State,Internal),          %  mapped to an internal part
   state_eqn(_Part,Internal := NextState),    %  and the internal state is
                                              %  a function of both the
   derive_behavior(Module,NextState,Beh),     %  inputs and previous state
   substitute_state(Module,Beh,Next_State).

/*********************************************************/
/*                                                       */
/*                  substitute_state                     */
/*                                                       */
/*  This clause uses replace_all to replace occurrences  */
/*  of internal variables with the appropriate external  */
/*  black-box variable obtained by the state_map fact.   */
/*********************************************************/

substitute_state(Module,DerBeh,SubBeh) :-
   state_map(Module,External,Internal),
   !,
   writeln([nl,'For module ',Module,':']),
   writeln['    Substituting: ', External,' for: ',Internal]),
   writeln([' Derived Behavior:           ',DerBeh]),
   replace_all(Module,Internal,External,DerBeh,SubBeh),
   writeln([' New (Substituted) Behavior:  ',SubBeh]).

/*********************************************************/
```

```
/*                                                          */
/*                  replace_all                             */
/*   Replaces each occurrence of an internal variable or    */
/*   other variables connected to this internal variable    */
/*   with the appropriate external black-box variable.      */
/*   The replace clauses allow you to traverse any type     */
/*   of behavioral structure and replace the appropriate    */
/*   variable name.  NOTE: New replace clauses will need    */
/*   to be added with new behavioral structures.            */
/************************************************************/

replace_all(Module,Old,New,OldBeh,SubBeh) :-
   replace(Old,New,OldBeh,SB),
   (   connected(Module,Old,Other) ;
       output_eqn(_Part,Other := Old) ),
   !,
   replace_all(Module,Other,New,SB,NewSB),
   evaluate1(NewSB,SubBeh).                    % Try to simplify further,
                                               %  if possible.
replace_all(_Module,_Old,_New,Beh,Beh) :-
   !.                                          % Module has no more connections!

replace(_Old,_New,Other,Other) :-
   atomic(Other),
   (not flag(terse) ->
      writeln(['Replace Rule2 -- ',Other,' is atomic'])
   |
    true),
   !.

replace(_Old,_New,Other,Other) :-
   var(Other),
   (not flag(terse) ->
      writeln(['Replace Rule3 -- ',Other,' is a variable'])
   |
    true),
   !.

replace(Old,_New,Other,Other) :-
   Old =.. [F,_Arg1],                  % keeps in(X) = in(incA(X))
   Other =.. [G,Arg2],                 %  from occuring, occurs test
   F \== G,
   ( var(Arg2)
   |
     atomic(Arg2) ),                   % this is already simplified
```

```
   (not flag(terse) ->
      writeln(['Replace Rule4 -- ',Other,' has variable/atomic argument'])
   |
    true),
   !.

replace(Old,New,and(X,Y),and(NewB1,NewB2)) :-
   !,(not flag(terse) ->
      writeln(['Replace Rule ''and'' '])
   |
    true),
   replace(Old,New,X,NewB1),
   replace(Old,New,Y,NewB2).

replace(Old,New,or(X,Y),or(NewB1,NewB2)) :-
   !,(not flag(terse) ->
      writeln(['Replace Rule ''or'' '])
   |
    true),
   replace(Old,New,X,NewB1),
   replace(Old,New,Y,NewB2).

replace(Old,New,neg(X),neg(NewB)) :-
   !,
   (not flag(terse) ->
      writeln(['Replace Rule ''neg'' '])
   |
    true),
   replace(Old,New,X,NewB).

replace(Old,New,X + Y,NewB1 + NewB2) :-
   !,
   (not flag(terse) ->
       writeln(['Replace Rule ''+'' '])
   |
    true),
   replace(Old,New,X,NewB1),
   replace(Old,New,Y,NewB2).

replace(Old,New,if(Cond,Texp,Fexp),if(NewB1,NewB2,NewB3)) :-
   !,
   (not flag(terse) ->
     writeln(['Replace Rule ''if'' '])
   |
    true),
```

```
   replace(Old,New,Cond,NewB1),
   replace(Old,New,Texp,NewB2),
   replace(Old,New,Fexp,NewB3).

replace(Old,New,Other,NewB) :-                % in(X) /== in(incA(X))
   Old =.. [F,Arg1],
   Other =.. [F,Arg2],
   (  var(Arg1),
      not var(Arg2)          % only one can be var
   |
      not var(Arg1),
      var(Arg2)
   ),
   replace(Old,New,Arg2,NewArgs),
   NewB =.. [F,NewArgs],                      %  Old behavior, or Old behavior
   (not flag(terse) ->
     writeln(['Replace Rule ''structure'' '])
    |
    true),
   !.                                         %   is some other nested structurer

replace(Old,New,Old,New) :-                    % If you find X replace with Y
   (not flag(terse) ->
      writeln(['Replace Rule 1 -- replace ',Old,' with ',New])
    |
    true),
    !.

replace(_Old,_New,Other,Other) :-
    (not flag(terse) ->
      writeln(['Default Rule'])
   |
    true).                % Default replace rule.
```

A.1.5 eqbeh.pl

```
/*************************************************************/
/*            EQBEH.PL    Equal_Behaviors                    */
/*  This file contains procedures necessary to determine     */
/*  the equivalence of a derived behavior(next state) and    */
/*  specified behavior(next state).                          */
/*  The equal_behaviors and equal_states clauses are used    */
/*  by derive_and_equate_behaviors and derive_and_equate_    */
/*  states to provide for every output/state equivalence.    */
/*  The primary methods of equivalence determination used    */
/*  are simiplification and boolean expansion.  The eqb      */
/*  clauses may expansion in future work.                    */
/*                                                           */
/*************************************************************/

equal_behaviors(Module,Output,Derived_Beh) :-
   output_eqn(Module,Output := Specified_Beh), % get specified behavior
   eqb(Module,Derived_Beh,Specified_Beh).

equal_states(Module,Nextstate,Derived_State) :-
   state_eqn(Module,Nextstate := Function),    % get specified state
   eqb(Module,Derived_State,Function).

/*************************************************************/
/*                  TRIVIAL IDENTITY                         */
/*************************************************************/

eqb(_M,X,X)  :-
   !.

/*************************************************************/
/*                BOOLEAN EXPANSION                          */
/*  The clause eq(NewDB,NewSB) is the driver for the code    */
/*  found in boole2.pro which performs CPT Dukes boolean     */
/*  expansion.                                               */
/*************************************************************/

eqb(_M,DB,SB) :-
 /* expandable(M),  fewer than to-be-determined combinations */
                              /* and boolean variables      */
   evaluate_dukes(DB,NewDB),
   evaluate_dukes(SB,NewSB),
   nl,
   writeln(['Does ',NewDB,' =']),
   writeln(['     ',NewSB,' ???',nl]),
```

```
   eq(NewDB,NewSB),
   writeln([' ',DB,' =']),
   writeln(['      ',SB]),
   writeln(['   By Boolean Expansion',nl]),
   !.

/*
expandable(M) :-                                % some how_many function will
   port(M,_,_,boole),                           %  be required
   !.                                           % Not currently used
*/

/*************************************************************/
/*                       SIMPLIFICATION                      */
/*************************************************************/

eqb(M,DB,SB) :-
   evaluate1(DB,NDB),
   evaluate1(SB,NSB),
   ( DB \== NDB ;
     SB \== NSB ),
   writeln(['Derived behavior is: ',DB]),
   eqb(M,NDB,NSB), !,
   nl.
```

A.1.6 eval.pl

```
/*************************************************************/
/*              EVAL.PL    Evaluate                          */
/*   This file performs a rudimentary simplification and    */
/*   and canonicalization on behavioral structures.         */
/*   Any new behavioral structures added will require       */
/*   additional evaluate_brown clauses.                     */
/*                                                          */
/*************************************************************/

evaluate1(X,EX) :-
   evaluate_brown(X,EX),
   (not flag(terse) ->
        ( writeln(['Value of ',X,':']),
          ((X==EX) ->
               writeln(['         is already canonical.'])
           |
           otherwise ->
               writeln(['         ',EX,nl])))
   |
    ((X\==EX) ->
  ( writeln(['Value of ',X,':']),
           writeln(['         ',EX,nl]))
    |
    true)).

/*************************************************************/
/*                                                          */
/*  The first three clauses provide basis cases for a       */
/*  behavioral structure, namely a Variable, Atom, or an    */
/*  elementary structure.                                   */
/*                                                          */
/*************************************************************/

evaluate_brown(X,X) :-
   var(X),
   !.

evaluate_brown(X,X) :-
   atomic(X),
   !.

evaluate_brown(Struct,Struct) :-
   Struct =.. [_F,Arg],
   ( var(Arg)
```

```
   |
     atom(Arg)
   ),
   !.

/*************************************************************/
/*                                                           */
/*  The next five clauses provide a method of simplifying    */
/*  and canonicalizing the boolean functions and, or, nor,   */
/*  nand, and negation. Another canonical form may prove     */
/*  quicker and these clauses would need to be modified      */
/*  accordingly.                                             */
/*                                                           */
/*  NOTE:  neg was originally chosen as the negation         */
/*  functor, as opposed to the more widely used not, since   */
/*  Prolog-1 defines not as the absence of a fact.  The      */
/*  functor not/1 isn't provided in Pure Prolog and thus     */
/*  does not exist in Quintus Prolog.  Since the program     */
/*  was moved to Quintus Prolog, not is now defined in the   */
/*  file qops.pl, along with other utility predicates.       */
/*************************************************************/

evaluate_brown(and(X,Y),Value) :-      % To evaluate and(X,Y), we evaluate
  evaluate_brown(X,EX),                %  both X and Y, and then evaluate
  evaluate_brown(Y,EY),                %  the 'and' operation
  (  (EX = 0
      |
      EY = 0),
     !,
     Value = 0
  |
     EX = 1,
     !,
     Value = EY
  |
     EY = 1,
     !,
     Value = EX
  |
     !,
     Value = and(EX,EY)
  ).

evaluate_brown(or(X,Y),Value) :-
   evaluate_brown(X,EX),
```

```
   evaluate_brown(Y,EY),
   (  (EX = 1
       |
       EY = 1),
      !,
      Value = 1
     |
      EX = 0,
      !,
      Value = EY
     |
      EY = 0,
      !,
      Value = EX
     |
      !,
      Value = or(EX,EY)
   ).

evaluate_brown(neg(X),Value)  :-
   evaluate_brown(X,EX),
   (  var(EX),
      !,
      Value = neg(X)
     |
      EX = 0,
      !,
      Value = 1
     |
      EX = 1,
      !,
      Value = 0
     |
      atom(EX),
      !,
      Value = neg(EX)
     |
      EX = neg(N),
      !,
      Value = N
     |
      EX = and(A1,A2),
      !,
      evaluate_brown(neg(A1),NA1),
      evaluate_brown(neg(A2),NA2),
```

```
      Value = or(NA1,NA2)
     |
      EX = or(O1,O2),
      !,
      evaluate_brown(neg(O1),NO1),
      evaluate_brown(neg(O2),NO2),
      Value = and(NO1,NO2)
     |
      !,
      Value = neg(EX)
   ).

/*************************************************************/
/*                                                           */
/*  The remaining clauses provide a method of simplifying    */
/*  and canonicalizing the functions required to verify      */
/*  the fulladder and counter, namely, if and +.             */
/*                                                           */
/*************************************************************/

evaluate_brown(if(Cond,Texp,Fexp),Value) :-
   evaluate_brown(Cond,NCond),
   evaluate_brown(Texp,NTexp),
   evaluate_brown(Fexp,NFexp),
   (  ( NCond = 1,
        !,                              % Condition is true
        Value = NTexp )                 %  return True exp
      |
      ( NCond = 0,
        !,                             % If False then
        Value = NFexp )                %  return False exp
      |
      ( NTexp = NFexp,
        !,                             % Condition is irrelevant
        Value = NTexp )                %  if choices equal
      |
      Value = if(NCond,NTexp,NFexp),   % otherwise return
      !                                %  simplified expression.
   ).

evaluate_brown(X+Y,Z) :-
   integer(X),
   integer(Y),
   !,
   Z is X + Y.                        % force simplification of 1 + 2 = 3
```

```
evaluate_brown(X+Y,Z) :-
   integer(Y),                          % X not integer due to cut in previous
   !,                                   %  clause
   evaluate_brown(X,NewX),
   Z = Y + NewX.                        % canonicalize with integer first

evaluate_brown(X+Y,Z) :-
   !,
   evaluate_brown(X,NewX),
   evaluate_brown(Y,NewY),
   Z = NewX + NewY.

evaluate_brown(X,X).                    %  default simplification for complex
                                        %  structures like in(incA(X)).
```

A.1.7 multdyn.pl

```
/*************************************************************/
/*                     MULTDYN.PL                            */
/*  This file sets the various component-related predicates */
/*  as being both multifile and dynamic.  Multifile/1 allows*/
/*  the definitions to be spread across a set of            */
/*  hierarchically consulted files, and dynamic/1 allows    */
/*  assert/1 and retract/1 to be used on these predicates   */
/*  by the executing AFIT_VERIFY system.  In order to allow */
/*  multifile/1 to function as we desire, this file _must_  */
/*  be reconsulted every time a verification is begun.      */
/*                                                          */
/*************************************************************/

:- multifile module_name/1, port/4, part/3, output_eqn/2, state_eqn/2,
   state_map/3, state_of/3, connected/3.

:- dynamic module_name/1, port/4, part/3, output_eqn/2, state_eqn/2,
   state_map/3, state_of/3, connected/3.


/* get_top(ComponentFile) clears out all 'old' module_name/1, port/4, */
/*  part/3, output_eqn/2, state_eqn/2, state_map/3, state_of/3,       */
/*  and connected/3 clauses before consulting the desired root file   */
/*  for the top-level component.  We check to make sure that the      */
/*  ComponentFile can be found in one of the library directories.     */

get_top(ComponentFile) :-
    check_for_read(ComponentFile),
    retractall(module_name(_)),   % if yes, then....
    retractall(port(_,_,_,_)),    % retract all these dynamic clauses
    retractall(part(_,_,_)),
    retractall(output_eqn(_,_)),
    retractall(state_eqn(_,_)),
    retractall(state_map(_,_)),
    retractall(state_of(_,_,_)),
    retractall(connected(_,_,_)),
    retractall(flag(loaded,_)),
    no_style_check(discontiguous),  % Turn off checking for procedures whose
                                    %  clauses are not all adjacent to one
                                    %  another in the file.
    no_style_check(multiple),       % Turn off checking for multiple
                                    %  definitions of the same procedure in
                                    %  different files.
    consult(library(ComponentFile)),
```

```
    style_check(all).

get_top(_ComponentFile) :-
    do_verify.
```

A.1.8 opentail.pl

```
/*************************************************************/
/*                      OPENTAIL.PL                          */
/*  This file contains various utilities for operating       */
/*  on lists with variable (open) tails.                     */
/*************************************************************/

/* add_to_opentailset(OpenTailList,NewElement) will return OpenTailList */
/* with the variable at its tail instantiated to [NewElement|NewVar],   */
/* as long as NewElement was not already an element of OpenTailList.    */

add_to_opentailset([NewElement|_Tail],NewElement) :-
    !.

add_to_opentailset([_Head|Tail],NewElement) :-
    var(Tail),
    !,
    Tail=[NewElement|_NewTail].

add_to_opentailset([_Head|Tail],NewElement) :-
    nonvar(Tail),
    !,
    add_to_opentailset(Tail,NewElement).

/* convert_to_notail(OpenTailList,ClosedList) converts a list with   */
/* an open (variable) tail to a standard closed list                 */

convert_to_notail([Head|Tail],[Head|NewTail]) :-
    nonvar(Tail),
    !,
    convert_to_notail(Tail,NewTail).

convert_to_notail([Head|Tail],[Head|[]]) :-
    var(Tail).

/* closed_list_to_string(List,String) */

closed_flatlist_to_string([],'[]') :-
    !.

closed_flatlist_to_string(List,String) :-
    closed_to_string_aux(List,'[',String).

closed_to_string_aux([Head|[]],Acc,String) :-
    string_append(Acc,Head,NewString),
```

```
    string_append(NewString,']',String).

closed_to_string_aux([Head|Tail],Acc,String) :-
    string_append(Acc,Head,Temp),
    string_append(Temp,',',NewString),
    closed_to_string_aux(Tail,NewString,String).
```

A.1.9 qops.pl

```
/************************************************************ */
/*                          QOPS.PL                          */
/*  This file provides the utility and operator definitions */
/*  to run verify.pl  on Quintus Prolog.  The :- multifile  */
/*  definition was required to allow Modules to be declared */
/*  in separate .pl  files.  Without this definition, any   */
/*  new module loaded would wipe out the previously loaded  */
/*  modules, and in the case of a fulladder (multiple file) */
/*  this was a problem (The defn of nand2 and xor was gone) */
/*  Operator precedence in Quintus and also pure Prolog     */
/*  goes from 0-1200, but Prolog-1 runs 0-255 (Prolog-1     */
/*  refman 5.5, Bratko p.182).                              */
/*                                                          */
/*  This file also provides the procedure definitions which */
/*  access the file system through the Quintus stream-based */
/*  file operations.  These procedures are discussed below. */
/************************************************************/

/* Be sure that the following Quintus libraries are loaded: */

:- ensure_loaded(library(readconst)).
:- ensure_loaded(library(strings)).
:- ensure_loaded(library(prompt)).
:- ensure_loaded(library(ask)).
:- ensure_loaded(library(basics)).
:- ensure_loaded(library(files)).


/*  make sure that flag/1 is defined as a dynamic predicate */

:- dynamic flag/1.


/*--------------------- UTILITIES ------------------------- */

writeln([]) :-
   nl.

writeln([nl|Rest]) :-          % provide a means of specifying a nl
   !,
   nl,
   writeln(Rest).

writeln([X|Rest]) :-
```

```
    !,
    write(X),
    writeln(Rest).

writeln([_]).

/*--------------------- OPERATORS ------------------------ */

:- unknown(Unknowns,fail).  /* Modified from  (trace,fail)        */

/*  Kevin Sparks's implementation of not/1 for Quintus Prolog     */
/* Removed in order to use a Quintus-derived implementation of not */
/*    ?- op(100, fy, not).     */
/*    not X :-                 */
/*        X,!,fail             */
/*        ;                    */
/*        true.                */

/* Definition of not/1 for Quintus Prolog.                         */
/*   Note that this does not check for free variables.  Use Quintus */
/*   Library not.pl for not/1 if this is necessary.                */

:- op(900,fy,not).            % Same as \+ builtin predicate

not(Goal) :-
    \+ call(Goal).


:- op(900, xfx, :=).
:- op(800, fx, if).



/*--------------------- FILE OPERATIONS ------------------- */

/*************************************************************************/
/* This contains various predicates which use the Quintus stream-based    */
/* file operations to read, write, and append information to files.  See   */
/* Quintus Prolog Reference Manual sections 5-1-* for more info on these   */
/* operations.                                                             */
/*    get_verified_parts     opens the file modfiles.list and reads        */
/*                              and returns the open list of module file   */
/*                              names                                      */
/*    save_verified_parts    opens the file modfiles.list and writes       */
/*                              the open list of verified module files     */
```

```
/*    load_known_parts       opens the file verified.parts and loads       */
/*                              the flag(verified(partname)) terms into    */
/*                              working memory                             */
/*    list_known_parts       writes each part with flag(verified(part))    */
/*                              to screen                                  */
/*    read_in_ver(Stream)    helper procedure that reads terms from file   */
/*                               Stream, asserting them as                 */
/*                               flag(verified(parts))until EOF is reached */
/*    update_known_parts     opens the file verified.parts and adds any    */
/*                              additional flag(verified(partname))        */
/*                              clauses into this file                     */
/*    write_parts(Stream)    helper procedure that writes                  */
/*                              flag(verified(part) clauses from working   */
/*                              memory into Stream                         */
/*    copy new_module(ModuleFileName)  uses UNIX cp command to move a      */
/*                              copy of ModuleFileName into the Parts      */
/*                              library                                    */
/*    save_new_module(ModuleFileName)  opens the file ModuleFileName and   */
/*                              creates a copy in the Parts library        */
/*    load_in(FileName)      used in a component file to load (if          */
/*                              necessary) any subordinate component files.*/
/*    extract_old_module(ModuleFileName) uses UNIX cp command to move a    */
/*                              copy of ModuleFileName from the Parts      */
/*                              directory to the user's current directory  */
/*    check_for_read(TopFileName) checks if TopFileName can be located      */
/*                              for read access along the library path     */
/*************************************************************************/


get_verified_parts(PartsOpenList) :-
    open(library('modfiles.list'),read,MFile),
    read(MFile,PartsOpenList),
    close(MFile).

save_verified_parts(PartsOpenList) :-
    open(library('modfiles.list'),write,MFile),
    write_canonical(MFile,PartsOpenList),
    write(MFile,'.'),
    nl(MFile),
    close(MFile).

load_known_parts :-
    open(library('parts.verified'),read, CFile),
    read_in_ver(CFile),
    close(CFile),
```

```
    asserta(flag(parts_loaded)),
    !,
    list_known_parts.

/* List_Known_Parts/0 lists each part that is in current working memory */

list_known_parts:-
    nl,
    not flag(verified(_X)),
    !,
    writeln(['No parts have been verified during this session.']).

list_known_parts :-
    flag(verified(X)),
    writeln(['The part "',X,
             '" has been previously verified during this session.']),
    fail
    |
    true.

/* Read_in_ver(Stream) reads a term from the given Stream, asserting the */
/*   term that is given by the Stream as a flag(verified(ATerm)) fact    */
/*   in current working memory.  This is a helper procedure used to      */
/*   preload verified parts information into working memory.             */

read_in_ver(Stream) :-
    read(Stream,ATerm),
    ( ATerm == end_of_file
    |
      ( (not flag(verified(ATerm)) ->
           asserta(flag(verified(ATerm)))
        |
         true),
        read_in_ver(Stream)
      )
    ).

/* Update_Known_Parts saves the current state of the flag(verified(X))  */
/*   state of working memory memory using flag(was_verified(X)) facts,  */
/*   adds in any previously stored flag(verified(X)) facts from a file  */
/*   using read_in_ver/1, stores the new list of flag(verified(X))      */
/*   facts back into a file, and resets the state of working memory to  */
/*   reflect the set of flag(verified(X)) facts that were saved.        */

update_known_parts :-
```

```
    (flag(verified(PartName)),
     asserta(flag(was_verified(PartName))),   % save current list
     fail                                     % of verified parts
    |
     true),
    open(library('parts.verified'),read, InFile),
    read_in_ver(InFile),
    close(InFile),
    open(library('parts.verified'),write, NewFile),
    write_parts(NewFile),
    close(NewFile),
    retractall(flag(verified(_PN))),        % clean up from file update
    (flag(was_verified(PartName2)),           % by restoring current
     asserta(flag(verified(PartName2))),      % list of verified parts
     fail
    |
     retractall(flag(was_verified(_PNs))) ).

/* Write_Parts(Stream) writes the names of all parts currently        */
/*   asserted in working memory as flag(verified(PartName)) facts     */
/*   to the current Stream.                                           */

write_parts(Stream) :-
    !,
    ( (flag(verified(X)),                   % find a verified part
       write_canonical(Stream,X),           % and write its name
       write(Stream,'.'),
       nl(Stream),
       fail)
    |
      true
    ).

/* Copy_New_Module(ModuleFileName) uses the UNIX cp command to copy a file */
/*       from its original directory into the Parts library directory.     */

copy_new_module(ModuleFileName) :-
    library_directory(Dir),
    substring(Dir,'Parts',_,_),                % get path to Parts directory
    string_append(Dir,'/',Dir2),
    string_append(Dir2,ModuleFileName,NewFN),
    string_append(NewFN,'.pl',NewFileName),     % construct new filename
    (file_exists(NewFileName,exists) ->
          writeln(['Sorry, but file ',ModuleFileName,
                   '.pl is already in the components library.'])
```

```
    |
     otherwise ->
     ( (string_append('./',ModuleFileName,RelFN),
        absolute_file_name(RelFN,AbsFileName),
        ( file_exists(AbsFileName,[read,exists]),
          string_append('cp ',AbsFileName,Cmnd1),   % construct cp command
          string_append(Cmnd1,' ',Cmnd2),
          string_append(Cmnd2,NewFileName,UnixCommand),
          unix(system(UnixCommand)),                % execute UNIX cp command
          get_verified_parts(PartsOpenList),        % update file listing
          add_to_opentailset(PartsOpenList,ModuleFileName),
          save_verified_parts(PartsOpenList),       %   of verified parts
          update_known_parts                        % update preverified
                                                    %   parts listing file
        )
       |
       writeln(['Sorry, but file ',ModuleFileName,
              '.pl can''t be found for read.'])
    ))).

/* Extract_Old_Module(ModuleFileName) uses the UNIX cp command to copy a   */
/*       file from the Parts library directory to the users current        */
/*       directory.                                                        */

extract_old_module(ModuleFileName) :-
    library_directory(Parts),
    substring(Parts,'Parts',_,_),           % get path to Parts directory
    string_append(Parts,'/',PartsSlash),
    string_append(PartsSlash,ModuleFileName,ModFN),
    absolute_file_name(ModFN,AbsModFN),
    (file_exists(AbsModFN) ->
      ( library_directory(Home),
        substring(Home,'.',_,_),
        absolute_file_name(Home,AbsHomeDir), % current user directory
        string_append('cp ',AbsModFN,Cmnd1),   % construct cp command
        string_append(Cmnd1,' ',Cmnd2),
        string_append(Cmnd2,AbsHomeDir,UnixCommand),
        unix(system(UnixCommand)),                % execute UNIX cp command
        writeln([nl,'Module ',ModuleFileName,
                 ' moved to current directory',nl]))
    |
     otherwise ->
        writeln([nl,'ERROR! ',ModuleFileName,' could not be found!'])
    ).
```

```
/* load_in(FileName) is used in a component file to load (if   */
/*  necessary) any subordinate component files.                */


load_in(FileName) :-
    ( not flag(loaded,FileName) ->
       (reconsult(library(FileName)),          % load one time, then
         asserta(flag(loaded,FileName))))      %  flag it
    |
    true.                                      % component already loaded


/* check_for_read(ComponentFile) checks if the given ComponentFile  */
/*   can be located along the current library path with read access */

check_for_read(ComponentFile) :-
    library_directory(LibDir),
    string_append(LibDir,'/',ThisDir),
    string_append(ThisDir,ComponentFile,RelFN),
    absolute_file_name(RelFN,AbsFileName),
    file_exists(AbsFileName,[read,exists]),
    !.

check_for_read(ComponentFile) :-
    writeln(['Sorry, but file ',ComponentFile,
              '.pl can''t be found for read.',nl]),
    do_verify.
```

A.2 Parts Library Listings

A.2.1 parts.verified

```
halfadd.
xor.
mux.
reg.
inc.
counter.
faddxor.
aoi.
nor2.
nand5.
nand4.
nand3.
inv.
nand2.
```

A.2.2 counter.pl

```
/*********************************************************/
/*                                                       */
/*  Counter.pl                                           */
/*                                                       */
/*     Module definitions for the counter example        */
/*     in Barrow's VERIFY article.                       */
/*                                                       */
/*********************************************************/

:- load_in(primitive).     %  get inc, reg, mux

/*--------------------  COUNTER ----------------------*/

:- retractall(module_name(_)).     % Make sure that THIS is the top module
module_name(counter).

port(counter,in(_ACounter),input,integer).
port(counter,ctrl(_ACounter),input,boole).
port(counter,out(_ACounter),output,integer).

part(counter,muxA(_ACounter),mux).
part(counter,regA(_ACounter),reg).
part(counter,incA(_ACounter),inc).

connected(counter,ctrl(ACounter),switch(muxA(ACounter))).
connected(counter,in(ACounter),in1(muxA(ACounter))).
connected(counter,out(muxA(ACounter)),in(regA(ACounter))).
connected(counter,out(regA(ACounter)),in(incA(ACounter))).
connected(counter,out(incA(ACounter)),in0(muxA(ACounter))).
connected(counter,out(regA(ACounter)),out(ACounter)).

/*  Behavior Specification */

state_of(counter,count(_ACounter),integer).
state_map(counter,count(ACounter),contents(regA(ACounter))).

output_eqn(counter,
                  out(ACounter) := count(ACounter) ).

state_eqn(counter,
                 count(ACounter) :=  if(ctrl(ACounter),
                                        in(ACounter),
                                        count(ACounter) + 1)).
```

A.2.3 faddxor.pl

```
/*************************************************************/
/*                    FADDXOR.PL                             */
/*  This file provides the specification of a fulladder      */
/*  composed of nand and exclusive or gates.  The file       */
/*  xor.pro must also be loaded to provide the module        */
/*  specifications for nand2 and xor.                        */
/*                                                           */
/*************************************************************/

:- load_in(primitive).     % insure nand2 is loaded
:- load_in(xor).     % insure xor is loaded


/* Structural specification for a full adder with xors   */

:- retractall(module_name(_)).     % Make sure that THIS is the top module
module_name(faddxor).

port(faddxor,x(_AFaddxor),input,boole).
port(faddxor,y(_AFaddxor),input,boole).
port(faddxor,cin(_AFaddxor),input,boole).
port(faddxor,outcarry(_AFaddxor),output,boole).
port(faddxor,outsum(_AFaddxor),output,boole).

part(faddxor,g1(_AFaddxor),nand2).
part(faddxor,g2(_AFaddxor),nand2).
part(faddxor,g3(_AFaddxor),nand2).
part(faddxor,g4(_AFaddxor),xor).
part(faddxor,g5(_AFaddxor),xor).

connected(faddxor,x(Afaddxor),in0(g1(Afaddxor))).
connected(faddxor,y(Afaddxor),in1(g1(Afaddxor))).
connected(faddxor,cin(Afaddxor),in0(g2(Afaddxor))).
connected(faddxor,out(g4(Afaddxor)),in1(g2(Afaddxor))).
connected(faddxor,out(g1(Afaddxor)),in0(g3(Afaddxor))).
connected(faddxor,out(g2(Afaddxor)),in1(g3(Afaddxor))).
connected(faddxor,x(Afaddxor),in0(g4(Afaddxor))).
connected(faddxor,y(Afaddxor),in1(g4(Afaddxor))).
connected(faddxor,out(g4(Afaddxor)),in0(g5(Afaddxor))).
connected(faddxor,cin(Afaddxor),in1(g5(Afaddxor))).
connected(faddxor,out(g5(Afaddxor)),outsum(Afaddxor)).
connected(faddxor,out(g3(Afaddxor)),outcarry(Afaddxor)).

/* Behavioral Specification          */
```

```
output_eqn(faddxor,
          outcarry(Afaddxor) :=
             or( and( x(Afaddxor),y(Afaddxor)),
                 and( cin(Afaddxor),
                      xor( x(Afaddxor),y(Afaddxor))  ))).
output_eqn(faddxor,
          outsum(Afaddxor) :=
            xor( xor(x(Afaddxor),y(Afaddxor)),
                 cin(Afaddxor) )).
```

A.2.4 modfiles.pl

```
'.'(xor,'.'(faddxor,'.'(counter,'.'(inv,'.'(aoi,'.'(halfadd,'.'(nand3,'.'(nand4,'.'(nand5
```

A.2.5 primitive.pl

```
/*************************************************************/
/*                  PRIMITIVES.PL                            */
/*  This file provides the module descriptions for           */
/*  a set of primitive components.  Primitive components     */
/*  are those components which do not use other components   */
/*  (that is, do not have "parts" clauses).  Components      */
/*  in this file are:  nor2, nand2, inc,reg,mux              */
/*                                                           */
/*************************************************************/

/*-----  Structural Specification for 2-input nor ----*/

module_name(nor2).

port(nor2,in0(_ANor2),input,boole).
port(nor2,in1(_ANor2),input,boole).
port(nor2,out(_ANor2),output,boole).

/* Behavioral Specification   */

output_eqn(nor2,
          out(ANor2) := and( neg(in0(ANor2)), neg(in1(ANor2))) ).

/*-----  Structural Specification for 2-input nand ----*/

module_name(nand2).

port(nand2,in0(_ANand2),input,boole).
port(nand2,in1(_ANand2),input,boole).
port(nand2,out(_ANand2),output,boole).

/* Behavioral Specification   */

output_eqn(nand2,
          out(ANand2) := or( neg(in0(ANand2)), neg(in1(ANand2))) ).


/*----------------- INCREMENTER ----------------------*/

module_name(inc).

port(inc,in(_AnInc),input,integer).
port(inc,out(_AnInc),output,integer).
```

```
/*  Behavior Specification */

output_eqn(inc,
              out(AnInc) := 1 + in(AnInc)).

/*----------------  MULTIPLEXER ----------------------*/

module_name(mux).

port(mux,in0(_AMux),input,integer).
port(mux,in1(_AMux),input,integer).
port(mux,switch(_AMux),input,boole).
port(mux,out(_AMux),output,integer).

/*  Behavior Specification */

output_eqn(mux,
              out(AMux) :=  if(switch(AMux),
                                  in1(AMux),in0(AMux))).

/*---------------------  REGISTER -------------------*/

module_name(reg).

port(reg,in(_AReg),input,integer).
port(reg,out(_AReg),output,integer).

/*  Behavior Specification */

state_of(reg,contents(_AReg),integer).

output_eqn(reg,
              out(AReg) := contents(AReg)).

state_eqn(reg,
             contents(AReg) := in(AReg)).
```

A.2.6 xor.pl

```
/*************************************************************/
/*                        XOR.PL                             */
/*  This file provides the module descriptions for           */
/*  exclusive ors.  This file is required when verifying a   */
/*  fulladder.  This file requires the loading of            */
/*  a set of primative components which includes a 2-input   */
/*  nand.                                                    */
/*                                                           */
/*************************************************************/

:- load_in(primitive).     % insure nand2 is loaded

/* Structural specification for a two-input exclusive or  */

:- retractall(module_name(_)).     % Make sure that THIS is the top module
module_name(xor).

port(xor,in0(_AnXor),input,boole).
port(xor,in1(_AnXor),input,boole).
port(xor,out(_AnXor),output,boole).

part(xor,g1(_AnXor),nand2).
part(xor,g2(_AnXor),nand2).
part(xor,g3(_AnXor),nand2).
part(xor,g4(_AnXor),nand2).

connected(xor,in0(AnXor),in0(g1(AnXor))).
connected(xor,in1(AnXor),in1(g1(AnXor))).
connected(xor,in0(AnXor),in0(g2(AnXor))).
connected(xor,out(g1(AnXor)),in1(g2(AnXor))).
connected(xor,out(g1(AnXor)),in0(g3(AnXor))).
connected(xor,in1(AnXor),in1(g3(AnXor))).
connected(xor,out(g2(AnXor)),in0(g4(AnXor))).
connected(xor,out(g3(AnXor)),in1(g4(AnXor))).
connected(xor,out(g4(AnXor)),out(AnXor)).

/* Behavioral Specification for a two-input XOR */

output_eqn(xor,
          out(AnXor) :=  or( and( neg(in0(AnXor)),
                                  in1(AnXor) ),
                             and( in0(AnXor),
                                  neg(in1(AnXor)) ))).
```

A.2.7 inv.pl

```
/*************************************************************/
/*                        INV.PL                             */
/*  This file provides the module descriptions for           */
/*  an inverter constructed from a 2-input nand.  This       */
/*  file requires the loading of a set of primitive          */
/*  components which includes a 2-input nand.                */
/*                                                           */
/*************************************************************/

:- load_in(primitive).     % insure nand2 is loaded

/* Structural specification for an inverter            */

:- retractall(module_name(_)).     % Make sure that THIS is the top module
module_name(inv).

port(inv,in(_AnInverter),input,boole).
port(inv,out(_AnInverter),output,boole).

part(inv,g1(_AnInverter),nand2).

connected(inv,in(AnInverter),in0(g1(AnInverter))).
connected(inv,in(AnInverter),in1(g1(AnInverter))).
connected(inv,out(g1(AnInverter)),out(AnInverter)).

/* Behavioral Specification for an inverter */

output_eqn(inv, out(AnInverter) := neg(in(AnInverter))).
```

A.2.8 aoi.pl

```
/*********************************************************/
/*                  And_Or_Invert                        */
/*                                                       */
/*  And-Or-Invert Module, as per Zycad library, pg 10-49 */
/*                                                       */
/*********************************************************/

:- load_in(primitive).     %  get nor2
:- load_in(inv).                    %  get inverter

/*---------------------- AOI ------------------------*/

:- retractall(module_name(_)).     % Make sure that THIS is the top module
module_name(aoi).

port(aoi,in0(_AnAOI),input,boole).
port(aoi,in1(_AnAOI),input,boole).
port(aoi,in2(_AnAOI),input,boole).
port(aoi,out(_AnAOI),output,boole).

part(aoi,nor2_1(_AnAOI),nor2).
part(aoi,nor2_2(_AnAOI),nor2).
part(aoi,inv_1(_AnAOI),inv).
part(aoi,inv_2(_AnAOI),inv).


connected(aoi,in0(AnAOI),in(inv_1(AnAOI))).
connected(aoi,in1(AnAOI),in(inv_2(AnAOI))).
connected(aoi,out(inv_1(AnAOI)),in0(nor2_1(AnAOI))).
connected(aoi,out(inv_2(AnAOI)),in1(nor2_1(AnAOI))).
connected(aoi,out(nor2_1(AnAOI)),in0(nor2_2(AnAOI))).
connected(aoi,in2(AnAOI),in1(nor2_2(AnAOI))).
connected(aoi,out(nor2_2(AnAOI)),out(AnAOI)).

/* Behavioral Specification         */

output_eqn(aoi, out(AnAOI) := neg( or( and( in0(AnAOI),
                                            in1(AnAOI)),
                                       in2(AnAOI) ) ) ).
```

A.2.9 halfadd.pl

```
/*****************************************************/
/*              HALFADD.PL                           */
/*                                                   */
/*  This file implements a simple half-adder that    */
/*  is built from inverters and 2 input nand gates.  */
/*  It is based upon a Zycad VHDL file written by    */
/*  Capt Dave Banton, which is attached below the    */
/*  Prolog code.                                     */
/*****************************************************/


:- load_in(primitive).     %  get nand2
:- load_in(inv).                    %  get inverter

/*---------------------- halfadd ------------------------*/

:- retractall(module_name(_)).   % Make sure that THIS is the top module
module_name(halfadd).

port(halfadd,in0(_HalfAdder),input,boole).       % bit 0 (low-order bit)
port(halfadd,in1(_HalfAdder),input,boole).       % bit 1 (high-order bit)
port(halfadd,sum(_HalfAdder),output,boole).
port(halfadd,carry(_HalfAdder),output,boole).

part(halfadd,inv_0(_HalfAdder),inv).
part(halfadd,inv_1(_HalfAdder),inv).
part(halfadd,inv_2(_HalfAdder),inv).
part(halfadd,nand2_0(_HalfAdder),nand2).
part(halfadd,nand2_1(_HalfAdder),nand2).
part(halfadd,nand2_2(_HalfAdder),nand2).
part(halfadd,nand2_3(_HalfAdder),nand2).

connected(halfadd,in0(HalfAddNand),in(inv_0(HalfAddNand))).
connected(halfadd,out(inv_0(HalfAddNand)),in1(nand2_1(HalfAddNand))).

connected(halfadd,in1(HalfAddNand),in(inv_1(HalfAddNand))).
connected(halfadd,out(inv_1(HalfAddNand)),in1(nand2_0(HalfAddNand))).

connected(halfadd,in0(HalfAddNand),in0(nand2_0(HalfAddNand))).
connected(halfadd,out(nand2_0(HalfAddNand)),in0(nand2_2(HalfAddNand))).

connected(halfadd,in1(HalfAddNand),in0(nand2_1(HalfAddNand))).
connected(halfadd,out(nand2_1(HalfAddNand)),in1(nand2_2(HalfAddNand))).
```

```
connected(halfadd,out(nand2_2(HalfAddNand)),sum(HalfAddNand)).

connected(halfadd,in0(HalfAddNand),in0(nand2_3(HalfAddNand))).
connected(halfadd,in1(HalfAddNand),in1(nand2_3(HalfAddNand))).
connected(halfadd,out(nand2_3(HalfAddNand)),in(inv_2(HalfAddNand))).
connected(halfadd,out(inv_2(HalfAddNand)),carry(HalfAddNand)).

/* Behavioral Specification for an half adder */

output_eqn(halfadd,
        sum(HalfAddNand) :=  or( and( neg( in0(HalfAddNand)),
                                      in1(HalfAddNand) ),
                                 and( in0(HalfAddNand),
                                      neg( in1(HalfAddNand)) )) ).

output_eqn(halfadd,
        carry(HalfAddNand) :=  and( in0(HalfAddNand),
                                    in1(HalfAddNand)) ).


/*****************************************************************
--- Adapted from Zycad VHDL File:
-----------------------------------------------------------------
-- DATE: 23 May 1991
-- VERSION:  1
-- UNIX FILENAME:  half_adder.vhalfadd
-- FUNCTION: This file is a structural description of a half_adder.
-- AUTHOR: dwb  (Capt David Banton)
-----------------------------------------------------------------
library ZYCAD;
use     ZYCAD.TYPES.all;
use     ZYCAD.COMPONENTS.all;
--
--  THE ENTITY DECLARATION:
--
entity half_adder is                    -- half adder
  port(Input: In MVL7_Vector (1 to 2); -- input
       Sum,                           -- sum
       Carry: Out MVL7);              -- carry
end half_adder;
--
-- THE ARCHITECTURAL BODY:
--
architecture Structural of half_adder is
--
```

```
signal S1, S2, S3, S4, S5:  MVL7;
--
component invgate                          -- inverter
  generic (tLH: Time;                      -- rise inertial delay
           tHL: Time);                     -- fall inertial delay
  port (Input:  In  MVL7;                  -- input
        Output: Out MVL7);                 -- output
end component;
--
component nandgate                         -- N input NAND gate
  generic (N: Positive;                    -- number of inputs
           tLH: Time;                      -- rise inertial delay
           tHL: Time);                     -- fall inertial delay
  port (Input:  In  MVL7_Vector (1 to N); -- input
        Output: Out MVL7);                 -- output
end component;
--
begin
 U1:  invgate  generic map (1 ns, 1 ns)
               port map (Input(1), S1);
 U2:  invgate  generic map (1 ns, 1 ns)
               port map (Input(2), S2);
 U3:  nandgate generic map (2, 2 ns, 2 ns)
               port map (Input(1) => Input(1), Input(2) => S2,
                         Output   => S3);
 U4:  nandgate generic map (2, 2 ns, 2 ns)
               port map (Input(1) => Input(2), Input(2) => S1,
                         Output   => S4);
 U5:  nandgate generic map (2, 2 ns, 2 ns)
               port map (Input(1) => S3, Input (2) => S4,
                         Output   => Sum);
 U6:  nandgate generic map (2, 2 ns, 2 ns)
               port map (Input(1) => Input(1), Input(2) => Input(2),
                         Output   => S5);
 U7:  invgate  generic map (1 ns, 1 ns)
               port map (S5, Carry);
--
end Structural;
-------------------------------------------------------------
-- FUNCTION: This file contains the test bench for
--           half_adder.vhalfadd.
-- AUTHOR: dwb  (Capt David Banton)
-------------------------------------------------------------
library ZYCAD;
use     ZYCAD.TYPES.all;
```

```
use     ZYCAD.COMPONENTS.all;
--
--  THE ENTITY DECLARATION:
entity half_adder_test_bench is
end    half_adder_test_bench;
--
--  THE ARCHITECTURAL BODY:
architecture test of half_adder_test_bench is
--
component half_adder
    port(Input: In  MVL7_Vector (1 to 2);
         Sum,
         Carry: Out MVL7);
end component;
--
signal Input: MVL7_Vector (1 to 2);
signal Sum, Carry: MVL7;
signal stop_sim: boolean := FALSE;
--
begin
  Problem: half_adder port map (Input, Sum, Carry);
--
  Input     <=   "00",              "01" after  50 ns,
                 "10" after 100 ns, "11" after 150 ns;
--
  stop_sim <= TRUE after 200 ns;
--
STOP_CONTROL: process
  begin
  wait until stop_sim = TRUE;
  assert false report "Simulation Done" severity failure;
end process STOP_CONTROL;
end test;
-------------------------------------------------------------------------
-- FUNCTION: This is the configuration specification file for
--           half_adder.vhalfadd.
-- AUTHOR: dwb  (Capt David Banton)
-------------------------------------------------------------------------
--  THE CONFIGURATION DECLARATION:
--
configuration half_adder_system of half_adder_test_bench is
  for test
  end for;
end half_adder_system;
*************************************************************/
```

A.2.10 nand3.pl

```
/*********************************************************/
/*                     NAND3.PL                          */
/*                                                       */
/*********************************************************/

:- load_in(primitive).              %  get nand2
:- load_in(inv).                    %  get inverter

/*---------------------- NAND3 ------------------------*/

:- retractall(module_name(_)).     % Make sure that THIS is the top module
module_name(nand3).

port(nand3,in0(_NAND3),input,boole).
port(nand3,in1(_NAND3),input,boole).
port(nand3,in2(_NAND3),input,boole).
port(nand3,out(_NAND3),output,boole).

part(nand3,nand2_1(_NAND3),nand2).
part(nand3,nand2_2(_NAND3),nand2).
part(nand3,inv0(_NAND3),inv).

connected(nand3,in0(TheNAND3),in0(nand2_1(TheNAND3))).
connected(nand3,in1(TheNAND3),in1(nand2_1(TheNAND3))).
connected(nand3,out(nand2_1(TheNAND3)),in(inv0(TheNAND3))).
connected(nand3,out(inv0(TheNAND3)),in0(nand2_2(TheNAND3))).
connected(nand3,in2(TheNAND3),in1(nand2_2(TheNAND3))).
connected(nand3,out(nand2_2(TheNAND3)),out(TheNAND3)).

/* Behavioral Specification         */

output_eqn(nand3, out(TheNAND3) := neg( and( and( in0(TheNAND3),
                                                  in1(TheNAND3)),
                                             in2(TheNAND3) ) ) ).
```

A.2.11 nand4.pl

```
/*********************************************************/
/*                      NAND4.PL                         */
/*                                                       */
/*********************************************************/

:- load_in(primitive).              %  get nand2
:- load_in(inv).                    %  get inverter

/*---------------------- NAND4 ------------------------*/

:- retractall(module_name(_)).      % Make sure that THIS is the top module
module_name(nand4).

port(nand4,in0(_NAND4),input,boole).
port(nand4,in1(_NAND4),input,boole).
port(nand4,in2(_NAND4),input,boole).
port(nand4,in3(_NAND4),input,boole).
port(nand4,out(_NAND4),output,boole).

part(nand4,nand2_1(_NAND4),nand2).
part(nand4,nand2_2(_NAND4),nand2).
part(nand4,nand2_3(_NAND4),nand2).
part(nand4,inv0(_NAND4),inv).
part(nand4,inv1(_NAND4),inv).

connected(nand4,in0(TheNAND4),in0(nand2_1(TheNAND4))).
connected(nand4,in1(TheNAND4),in1(nand2_1(TheNAND4))).
connected(nand4,out(nand2_1(TheNAND4)),in(inv0(TheNAND4))).
connected(nand4,out(inv0(TheNAND4)),in0(nand2_2(TheNAND4))).
connected(nand4,in2(TheNAND4),in0(nand2_3(TheNAND4))).
connected(nand4,in3(TheNAND4),in1(nand2_3(TheNAND4))).
connected(nand4,out(nand2_3(TheNAND4)),in(inv1(TheNAND4))).
connected(nand4,out(inv1(TheNAND4)),in1(nand2_2(TheNAND4))).
connected(nand4,out(nand2_2(TheNAND4)),out(TheNAND4)).

/* Behavioral Specification         */

output_eqn(nand4, out(TheNAND4) := neg( and( and( in0(TheNAND4),
                                                  in1(TheNAND4)),
                                             and( in2(TheNAND4),
                                                  in3(TheNAND4)) ))).
```

A.2.12 nand5.pl

```
/*********************************************************/
/*                    NAND5.PL                           */
/*                                                       */
/*********************************************************/

:- load_in(nand3).                  %  get nand3
:- load_in(inv).                    %  get inverter

/*---------------------- NAND5 ------------------------*/

:- retractall(module_name(_)).      % Make sure that THIS is the top module
module_name(nand5).

port(nand5,in0(_NAND5),input,boole).
port(nand5,in1(_NAND5),input,boole).
port(nand5,in2(_NAND5),input,boole).
port(nand5,in3(_NAND5),input,boole).
port(nand5,in4(_NAND5),input,boole).
port(nand5,out(_NAND5),output,boole).

part(nand5,nand3_1(_NAND5),nand3).
part(nand5,nand3_2(_NAND5),nand3).
part(nand5,inv0(_NAND5),inv).

connected(nand5,in0(TheNAND5),in0(nand3_1(TheNAND5))).
connected(nand5,in1(TheNAND5),in1(nand3_1(TheNAND5))).
connected(nand5,in2(TheNAND5),in2(nand3_1(TheNAND5))).
connected(nand5,out(nand3_1(TheNAND5)),in(inv0(TheNAND5))).
connected(nand5,out(inv0(TheNAND5)),in0(nand3_2(TheNAND5))).
connected(nand5,in3(TheNAND5),in1(nand3_2(TheNAND5))).
connected(nand5,in4(TheNAND5),in2(nand3_2(TheNAND5))).
connected(nand5,out(nand3_2(TheNAND5)),out(TheNAND5)).

/* Behavioral Specification         */

output_eqn(nand5, out(TheNAND5) := neg( and( and( in0(TheNAND5),
                                                  in1(TheNAND5)),
                                             and( in2(TheNAND5),
                                                  and( in3(TheNAND5),
                                                       in4(TheNAND5) ) )))).
```

A.2.13 mux_4x1.pl

```
/*******************************************************/
/*              MUX4x1.PL                              */
/*                                                     */
/*  This file implements a 4-to-1 multiplexor that     */
/*  is built from inverters and 2-to-1 multiplexors.   */
/*  It is based upon a Zycad VHDL file written by      */
/*  Capt Dave Banton, which is attached below the      */
/*  Prolog code.                                       */
/*******************************************************/


:- load_in(primitive).     %  get mux (2x1)
:- load_in(inv).                    %  get inverter

/*---------------------- MUX4x1 -------------------------*/

:- retractall(module_name(_)).  % Make sure that THIS is the top module
module_name(mux4x1).

port(mux4x1,in0(_Mux4),input,integer).
port(mux4x1,in1(_Mux4),input,integer).
port(mux4x1,in2(_Mux4),input,integer).
port(mux4x1,in3(_Mux4),input,integer).
port(mux4x1,sel0(_Mux4),input,boole).     % bit 0 (low-order)
port(mux4x1,sel1(_Mux4),input,boole).     % bit 1 (high order)
port(mux4x1,out(_Mux4),output,integer).

part(mux4x1,inv_1(_Mux4),inv).
part(mux4x1,inv_2(_Mux4),inv).
part(mux4x1,mux2_0(_Mux4),mux).
part(mux4x1,mux2_1(_Mux4),mux).
part(mux4x1,mux2_2(_Mux4),mux).
                              % registers added to aid in writing
                              %  the output equation (they have no
                              %  intrinsic effect upon the operation)
part(mux4x1,reg_0(_Mux4),reg).
part(mux4x1,reg_1(_Mux4),reg).
part(mux4x1,reg_2(_Mux4),reg).

connected(mux4x1,in0(Mux4x1),in0(mux2_0(Mux4x1))).
connected(mux4x1,in1(Mux4x1),in1(mux2_0(Mux4x1))).
connected(mux4x1,in2(Mux4x1),in0(mux2_1(Mux4x1))).
connected(mux4x1,in3(Mux4x1),in1(mux2_1(Mux4x1))).
connected(mux4x1,out(mux2_0(Mux4x1)),in(reg_0(Mux4x1))).
```

```
connected(mux4x1,out(reg_0(Mux4x1)),in0(mux2_2(Mux4x1))).
connected(mux4x1,out(mux2_1(Mux4x1)),in(reg_1(Mux4x1))).
connected(mux4x1,out(reg_1(Mux4x1)),in1(mux2_2(Mux4x1))).
connected(mux4x1,out(mux2_2(Mux4x1)),in(inv_1(Mux4x1))).
connected(mux4x1,sel0(Mux4x1),switch(mux2_0(Mux4x1))).
connected(mux4x1,sel0(Mux4x1),switch(mux2_1(Mux4x1))).
connected(mux4x1,sel1(Mux4x1),switch(mux2_2(Mux4x1))).
connected(mux4x1,out(inv_1(Mux4x1)),in(inv_2(Mux4x1))).
connected(mux4x1,out(inv_2(Mux4x1)),in(reg_2(Mux4x1))).
connected(mux4x1,out(reg_2(Mux4x1)),out(Mux4x1)).

/* Behavioral Specification         */

state_of(mux4x1,out_value(_Mux4),integer).
state_map(mux4x1,out_value(Mux4x1),contents(reg_2(Mux4x1))).

state_of(mux4x1,out_r1(_Mux4),integer).
state_map(mux4x1,out_r1(Mux4x1),contents(reg_1(Mux4x1))).

state_of(mux4x1,out_r0(_Mux4),integer).
state_map(mux4x1,out_r0(Mux4x1),contents(reg_0(Mux4x1))).

state_eqn(mux4x1,out_value(Mux4x1) := if(sel1(Mux4x1),
                                        out_r1(Mux4x1),
                                        out_r0(Mux4x1))).

state_eqn(mux4x1,out_r0(Mux4x1) := if(sel0(Mux4x1),
                                        in3(Mux4x1),in2(Mux4x1))).

state_eqn(mux4x1,out_r1(Mux4x1) := if(sel0(Mux4x1),
                                        in0(Mux4x1),in1(Mux4x1))).

output_eqn(mux4x1,
                  out(Mux4x1) := out_value(Mux4x1)).

/*****************************************************************
--- Adapted from Zycad VHDL File:
-- DATE:  31 July 1991
-- VERSION:  1
-- UNIX FILENAME:  mux4x1_entity.vhd
-- FUNCTION: This file contains the entity and structural
-- architecture of a 4x1 mux.
-- AUTHOR: dwb  (Capt David Banton)
-----------------------------------------------------------------
library ZYCAD;
```

```
use ZYCAD.TYPES.all;
use ZYCAD.COMPONENTS.all;
--
--  THE ENTITY DECLARATION:
--
entity mux4X1 is                                -- 4 by 1 multiplexer
  port(In0,                                     -- data input 0
       In1,                                     -- data input 1
       In2,                                     -- data input 2
       In3:    In  MVL7;                        -- data input 3
       Sel:    In  MVL7_Vector (1 downto 0);
                                    -- select input (?? => In??)
       Output: Out MVL7);                       -- output
end mux4X1;
--
--  THE ARCHITECTURAL BODY:
--
architecture Structural of mux4X1 is
--
signal S0, S1, Temp_Output, Outputnot: MVL7;
--
component invgate           -- inverter
  generic (tLH: Time;       -- rise inertial delay
           tHL: Time);      -- fall inertial delay
  port (Input:  In MVL7;    -- input
        Output: Out MVL7); -- output
end component;
--
component mux2X1            -- 2 by 1 multiplexer
  generic (tLH: Time;       -- rise inertial delay
           tHL: Time);      -- fall inertial delay
  port (In0,                -- data input 0
        In1,                -- data input 1
        Sel:    In  MVL7;  -- select input (0=>In0)
        Output: Out MVL7); -- output
end component;
--
begin
  U0:  mux2X1  generic map (1 ns, 1 ns)
               port map (In0, In1, Sel(0), S0);
  U1:  mux2X1  generic map (1 ns, 1 ns)
               port map (In2, In3, Sel(0), S1);
  U2:  mux2X1  generic map (1 ns, 1 ns)
               port map (S0, S1, Sel(1), Temp_Output);
  U3:  invgate generic map (1 ns, 1 ns)
```

```
              port map (Temp_Output, Outputnot);
  U4:  invgate generic map (1 ns, 1 ns)
              port map (Outputnot, Output);
end Structural;
*****************************************************************/
```

A.2.14 halfadd.pl

```
/*****************************************************/
/*               HALFADD.PL                          */
/*                                                   */
/*  This file implements a simple half-adder that    */
/*  is built from inverters and 2 input nand gates.  */
/*  It is based upon a Zycad VHDL file written by    */
/*  Capt Dave Banton, which is attached below the    */
/*  Prolog code.                                     */
/*****************************************************/


:- load_in(primitive).     % get nand2
:- load_in(inv).                   % get inverter

/*---------------------- halfadd ------------------------*/

:- retractall(module_name(_)).   % Make sure that THIS is the top module
module_name(halfadd).

port(halfadd,in0(_HalfAdder),input,boole).       % bit 0 (low-order bit)
port(halfadd,in1(_HalfAdder),input,boole).       % bit 1 (high-order bit)
port(halfadd,sum(_HalfAdder),output,boole).
port(halfadd,carry(_HalfAdder),output,boole).

part(halfadd,inv_0(_HalfAdder),inv).
part(halfadd,inv_1(_HalfAdder),inv).
part(halfadd,inv_2(_HalfAdder),inv).
part(halfadd,nand2_0(_HalfAdder),nand2).
part(halfadd,nand2_1(_HalfAdder),nand2).
part(halfadd,nand2_2(_HalfAdder),nand2).
part(halfadd,nand2_3(_HalfAdder),nand2).

connected(halfadd,in0(HalfAddNand),in(inv_0(HalfAddNand))).
connected(halfadd,out(inv_0(HalfAddNand)),in1(nand2_1(HalfAddNand))).

connected(halfadd,in1(HalfAddNand),in(inv_1(HalfAddNand))).
connected(halfadd,out(inv_1(HalfAddNand)),in1(nand2_0(HalfAddNand))).

connected(halfadd,in0(HalfAddNand),in0(nand2_0(HalfAddNand))).
connected(halfadd,out(nand2_0(HalfAddNand)),in0(nand2_2(HalfAddNand))).

connected(halfadd,in1(HalfAddNand),in0(nand2_1(HalfAddNand))).
connected(halfadd,out(nand2_1(HalfAddNand)),in1(nand2_2(HalfAddNand))).
```

```
connected(halfadd,out(nand2_2(HalfAddNand)),sum(HalfAddNand)).

connected(halfadd,in0(HalfAddNand),in0(nand2_3(HalfAddNand))).
connected(halfadd,in1(HalfAddNand),in1(nand2_3(HalfAddNand))).
connected(halfadd,out(nand2_3(HalfAddNand)),in(inv_2(HalfAddNand))).
connected(halfadd,out(inv_2(HalfAddNand)),carry(HalfAddNand)).

/* Behavioral Specification for an half adder */

output_eqn(halfadd,
        sum(HalfAddNand) :=  or( and( neg( in0(HalfAddNand)),
                                      in1(HalfAddNand) ),
                                 and( in0(HalfAddNand),
                                      neg( in1(HalfAddNand)) )) ).

output_eqn(halfadd,
        carry(HalfAddNand) :=  and( in0(HalfAddNand),
                                    in1(HalfAddNand)) ).


/*****************************************************************
--- Adapted from Zycad VHDL File:
-----------------------------------------------------------------
-- DATE: 23 May 1991
-- VERSION:  1
-- UNIX FILENAME:  half_adder.vhalfadd
-- FUNCTION: This file is a structural description of a half_adder.
-- AUTHOR: dwb  (Capt David Banton)
-----------------------------------------------------------------
library ZYCAD;
use     ZYCAD.TYPES.all;
use     ZYCAD.COMPONENTS.all;
--
--  THE ENTITY DECLARATION:
--
entity half_adder is                    -- half adder
  port(Input: In MVL7_Vector (1 to 2); -- input
       Sum,                           -- sum
       Carry: Out MVL7);              -- carry
end half_adder;
--
-- THE ARCHITECTURAL BODY:
--
architecture Structural of half_adder is
--
```

```
signal S1, S2, S3, S4, S5:  MVL7;
--
component invgate                          -- inverter
  generic (tLH: Time;                      -- rise inertial delay
           tHL: Time);                     -- fall inertial delay
  port (Input:  In  MVL7;                  -- input
        Output: Out MVL7);                 -- output
end component;
--
component nandgate                         -- N input NAND gate
  generic (N: Positive;                    -- number of inputs
           tLH: Time;                      -- rise inertial delay
           tHL: Time);                     -- fall inertial delay
  port (Input:  In  MVL7_Vector (1 to N); -- input
        Output: Out MVL7);                 -- output
end component;
--
begin
 U1:  invgate  generic map (1 ns, 1 ns)
               port map (Input(1), S1);
 U2:  invgate  generic map (1 ns, 1 ns)
               port map (Input(2), S2);
 U3:  nandgate generic map (2, 2 ns, 2 ns)
               port map (Input(1) => Input(1), Input(2) => S2,
                         Output   => S3);
 U4:  nandgate generic map (2, 2 ns, 2 ns)
               port map (Input(1) => Input(2), Input(2) => S1,
                         Output   => S4);
 U5:  nandgate generic map (2, 2 ns, 2 ns)
               port map (Input(1) => S3, Input (2) => S4,
                         Output   => Sum);
 U6:  nandgate generic map (2, 2 ns, 2 ns)
               port map (Input(1) => Input(1), Input(2) => Input(2),
                         Output   => S5);
 U7:  invgate  generic map (1 ns, 1 ns)
               port map (S5, Carry);
--
end Structural;
-------------------------------------------------------------
-- FUNCTION: This file contains the test bench for
--           half_adder.vhalfadd.
-- AUTHOR: dwb  (Capt David Banton)
-------------------------------------------------------------
library ZYCAD;
use     ZYCAD.TYPES.all;
```

```
use     ZYCAD.COMPONENTS.all;
--
--  THE ENTITY DECLARATION:
entity half_adder_test_bench is
end    half_adder_test_bench;
--
--  THE ARCHITECTURAL BODY:
architecture test of half_adder_test_bench is
--
component half_adder
    port(Input: In  MVL7_Vector (1 to 2);
         Sum,
         Carry: Out MVL7);
end component;
--
signal Input: MVL7_Vector (1 to 2);
signal Sum, Carry: MVL7;
signal stop_sim: boolean := FALSE;
--
begin
  Problem: half_adder port map (Input, Sum, Carry);
--
  Input     <=   "00",              "01" after  50 ns,
                 "10" after 100 ns, "11" after 150 ns;
--
  stop_sim <= TRUE after 200 ns;
--
STOP_CONTROL: process
  begin
  wait until stop_sim = TRUE;
  assert false report "Simulation Done" severity failure;
end process STOP_CONTROL;
end test;
-------------------------------------------------------------------------
-- FUNCTION: This is the configuration specification file for
--           half_adder.vhalfadd.
-- AUTHOR: dwb  (Capt David Banton)
-------------------------------------------------------------------------
--  THE CONFIGURATION DECLARATION:
--
configuration half_adder_system of half_adder_test_bench is
  for test
  end for;
end half_adder_system;
*************************************************************************/
```

A.2.15 faddnor.pl

```
/*******************************************************/
/*              FADDNOR.PL                             */
/*                                                     */
/*  This file implements a simple full-adder that      */
/*  is built from inverters, half-adders, and          */
/*  2 input nor gates.                                 */
/*  It is based upon a Zycad VHDL file written by      */
/*  Capt Dave Banton, which is attached below the      */
/*  Prolog code.                                       */
/*******************************************************/


:- load_in(primitive).     %  get nor2
:- load_in(inv).                    %  get inverter
:- load_in(halfadd).                %  get half adder

/*---------------------- FADDNOR ------------------------*/

:- retractall(module_name(_)).     % Make sure that THIS is the top module
module_name(faddnor).

port(faddnor,in0(_FullAdder),input,boole).          % bit 0 (low-order bit)
port(faddnor,in1(_FullAdder),input,boole).          % bit 1 (high-order bit)
port(faddnor,carryin(_FullAdder),input,boole).
port(faddnor,sum(_FullAdder),output,boole).
port(faddnor,carryout(_FullAdder),output,boole).

part(faddnor,inv_0(_FullAdder),inv).
part(faddnor,nor2_0(_FullAdder),nor2).
part(faddnor,hadder_0(_FullAdder),halfadd).
part(faddnor,hadder_1(_FullAdder),halfadd).

connected(faddnor,in0(FullAdderNor),in0(hadder_0(FullAdderNor))).
connected(faddnor,in1(FullAdderNor),in1(hadder_0(FullAdderNor))).
connected(faddnor,sum(hadder_0(FullAdderNor)),in0(hadder_1(FullAdderNor))).
connected(faddnor,carry(hadder_0(FullAdderNor)),in1(nor2_0(FullAdderNor))).

connected(faddnor,carryin(FullAdderNor),in1(hadder_1(FullAdderNor))).
connected(faddnor,sum(hadder_1(FullAdderNor)),sum(FullAdderNor)).
connected(faddnor,carry(hadder_1(FullAdderNor)),in0(nor2_0'FullAdderNor))).

connected(faddnor,out(nor2_0(FullAdderNor)),in(inv_0(FullAdderNor))).
connected(faddnor,out(inv_0(FullAdderNor)),carryout(FullAdderNor)).
```

```
/* Behavioral Specification for an full adder */

output_eqn(faddnor,sum(FullAdderNor) :=
                      or(
                         or( and(
                                 and( in0(FullAdderNor),
                                      in1(FullAdderNor)),
                                 carryin(FullAdderNor)),
                             and(
                                 and( in0(FullAdderNor),
                                      neg(in1(FullAdderNor))),
                                 neg( carryin(FullAdderNor))) ),
                         or( and(
                                 and( neg( in0(FullAdderNor)),
                                      in1(FullAdderNor)),
                                 neg( carryin(FullAdderNor))),
                             and(
                                 and( neg( in0(FullAdderNor)),
                                      neg( in1(FullAdderNor))),
                                 carryin(FullAdderNor)) ) ) ).

output_eqn(faddnor,carryout(FullAdderNor) :=
                               or( or( and( in0(FullAdderNor),
                                            in1(FullAdderNor)),
                                       and( in0(FullAdderNor),
                                            carryin(FullAdderNor)) ),
                                   and( in1(FullAdderNor),
                                        carryin(FullAdderNor)) ) ).
```

```
/*****************************************************************
--- Adapted from Zycad VHDL File:
-----------------------------------------------------------------
-- DATE:  23 May 1991
-- VERSION:  1
-- UNIX FILENAME:  full_adder.vhd
-- FUNCTION: This file is a structural description of a full_adder.
-- AUTHOR: dwb (Capt David W. Banton)
-----------------------------------------------------------------
library ZYCAD;
use     ZYCAD types.all;
use     ZYCAD.COMPONENTS.all;
```

```
--
--  THE ENTITY DECLARATION:
--
entity full_adder is
  port(Input: In  MVL7_Vector (1 to 3); -- In1, In2, CarryIn
       Sum, CarryOut: Out MVL7);
end full_Adder;
--
-- THE ARCHITECTURAL BODY:
--
architecture Structural of full_adder is
--
signal P1,G1, S1, S2: MVL7;
--
component invgate                          -- inverter
  generic (tLH: Time;                      -- rise inertial delay
           tHL: Time);                     -- fall inertial delay
  port (Input:  In MVL7;                   -- input
        Output: Out MVL7);                 -- output
end component;
--
component norgate                          -- N input NOR gate
  generic (N: Positive;                    -- number of inputs
           tLH: Time;                      -- rise inertial delay
           tHL: Time);                     -- fall inertial delay
  port (Input:  In  MVL7_Vector (1 to 2); -- input
        Output: Out MVL7);                 -- output
end component;
--
component half_adder
  port (Input: In MVL7_Vector (1 to 2); Sum, Carry: Out MVL7);
end component;
--
begin
  U0 :  half_adder port map (Input(1) => Input(1), Input(2) => Input(2),
                             Sum      => P1,       Carry    => G1);
  U1 :  half_adder port map (Input(1) => P1,       Input(2) => Input(3),
                             Sum      => Sum,      Carry    => S1);
  U2 :  norgate generic map (2, 2 ns, 2 ns)
                port map (Input(1) => S1,       Input(2) => G1,
                          Output   => S2);
  U3 :  invgate generic map (1 ns, 1 ns)
                port map (S2, CarryOut);
--
end Structural;
```

```
------------------------------------------------------------
-- FUNCTION: test_bench
-- AUTHOR: dwb
------------------------------------------------------------
library ZYCAD;
use     ZYCAD.types.all;
use     ZYCAD.COMPONENTS.all;
--
--  THE ENTITY DECLARATION:
entity full_adder_test_bench is
end    full_adder_test_bench;
--
--  THE ARCHITECTURAL BODY:
--
architecture test of full_adder_test_bench is
--
component full_adder
  port(Input: In  MVL7_Vector (1 to 3); -- In1, In2, CarryIn
       Sum, CarryOut: Out MVL7);
end component;
--
signal Input: MVL7_Vector (1 to 3);
signal Sum, CarryOut: MVL7;
signal stop_sim: boolean := FALSE;
--
begin
  Problem: full_adder port map (Input, Sum, CarryOut);
--
  Input     <=   "000",              "001" after  50 ns,
                 "010" after 100 ns, "011" after 150 ns,
                 "100" after 200 ns, "101" after 250 ns,
                 "110" after 300 ns, "111" after 350 ns;
--
  stop_sim <= TRUE after 500 ns;
--
STOP_CONTROL: process
  begin
  wait until stop_sim = TRUE;
  assert false report "Simulation Done" severity failure;
end process STOP_CONTROL;
end test;
------------------------------------------------------------------------
-- FUNCTION: configuration
-- AUTHOR:dbw
------------------------------------------------------------------------
```

```
--  THE CONFIGURATION DECLARATION:
--
configuration full_adder_system of full_adder_test_bench is
  for test
  end for;
end full_adder_system;

**************************************************************/
```

A.2.16 fadd4_cl.pl

```
/*****************************************************/
/*              FADD4_CL.PL                          */
/*                                                   */
/*  This file implements a four bit full-adder with  */
/*  carry lookahead.  This part is built with xors,  */
/*  inverters, half adders, and 2, 3, 4, and 5 input */
/*  nand gates.                                      */
/*  It is based upon a Zycad VHDL file written by    */
/*  Capt Dave Banton, which is attached below the    */
/*  Prolog code.                                     */
/*****************************************************/


:- load_in(primitive).     %  get nand2
:- load_in(halfadd).              %  get half adder
:- load_in(inv).                  %  get inverter
:- load_in(xor).                  %  get xor
:- load_in(nand3).                %  get 3 input nand
:- load_in(nand4).                %  get 4 input nand
:- load_in(nand5).                %  get 5 input nand

/*---------------------- 4bit adder ------------------------*/

:- retractall(module_name(_)).     % Make sure that THIS is the top module
module_name(fadd4_cl).

port(fadd4_cl,in00(_FullAdder),input,boole). % nibble 0 bit 0 (low-order bit)
port(fadd4_cl,in01(_FullAdder),input,boole). % nibble 0 bit 1
port(fadd4_cl,in02(_FullAdder),input,boole). % nibble 0 bit 2
port(fadd4_cl,in03(_FullAdder),input,boole). % nibble 0 bit 3 (high-order bit)

port(fadd4_cl,in10(_FullAdder),input,boole). % nibble 1 bit 0 (low-order bit)
port(fadd4_cl,in11(_FullAdder),input,boole). % nibble 1 bit 1
port(fadd4_cl,in12(_FullAdder),input,boole). % nibble 1 bit 2
port(fadd4_cl,in13(_FullAdder),input,boole). % nibble 1 bit 3 (high-order bit)

port(fadd4_cl,carryin(_FullAdder),input,boole).

port(fadd4_cl,sum0(_FullAdder),output,boole).
port(fadd4_cl,sum1(_FullAdder),output,boole).
port(fadd4_cl,sum2(_FullAdder),output,boole).
port(fadd4_cl,sum3(_FullAdder),output,boole).
port(fadd4_cl,carryout(_FullAdder),output,boole).
```

```
/*------------------ parts list--------------------------*/

part(fadd4_cl,u0(_FullAdder),halfadd).        % half adders
part(fadd4_cl,u1(_FullAdder),halfadd).
part(fadd4_cl,u2(_FullAdder),halfadd).
part(fadd4_cl,u3(_FullAdder),halfadd).

part(fadd4_cl,u4(_FullAdder),nand2).          % sum and carry 0
part(fadd4_cl,u5(_FullAdder),inv).
part(fadd4_cl,u6(_FullAdder),nand2).

part(fadd4_cl,u7(_FullAdder),nand2).          % sum and carry 1
part(fadd4_cl,u8(_FullAdder),nand3).
part(fadd4_cl,u9(_FullAdder),inv).
part(fadd4_cl,u10(_FullAdder),nand3).

part(fadd4_cl,u11(_FullAdder),nand2).         % sum and carry 2
part(fadd4_cl,u12(_FullAdder),nand3).
part(fadd4_cl,u13(_FullAdder),nand4).
part(fadd4_cl,u14(_FullAdder),inv).
part(fadd4_cl,u15(_FullAdder),nand4).

part(fadd4_cl,u16(_FullAdder),nand2).         % sum and carry 3
part(fadd4_cl,u17(_FullAdder),nand3).
part(fadd4_cl,u18(_FullAdder),nand4).
part(fadd4_cl,u19(_FullAdder),nand5).
part(fadd4_cl,u20(_FullAdder),inv).
part(fadd4_cl,u21(_FullAdder),nand5).

part(fadd4_cl,u22(_FullAdder),xor).           % generate sum bits
part(fadd4_cl,u23(_FullAdder),xor).
part(fadd4_cl,u24(_FullAdder),xor).
part(fadd4_cl,u25(_FullAdder),xor).


connected(fadd4_cl,in00(FullAdd4CL),in0(u0(FullAdd4CL))).
connected(fadd4_cl,in10(FullAdd4CL),in1(u0(FullAdd4CL))).
connected(fadd4_cl,in01(FullAdd4CL),in0(u1(FullAdd4CL))).
connected(fadd4_cl,in11(FullAdd4CL),in1(u1(FullAdd4CL))).
connected(fadd4_cl,in02(FullAdd4CL),in0(u2(FullAdd4CL))).
connected(fadd4_cl,in12(FullAdd4CL),in1(u2(FullAdd4CL))).
connected(fadd4_cl,in03(FullAdd4CL),in0(u2(FullAdd4CL))).
connected(fadd4_cl,in13(FullAdd4CL),in1(u2(FullAdd4CL))).

connected(fadd4_cl,carryin(FullAdd4CL),in0(u4(FullAdd4CL))).
```

```
connected(fadd4_cl,sum(u0(FullAdd4CL)),in1(u4(FullAdd4CL))).
connected(fadd4_cl,carry(u0(FullAdd4CL)),in(u5(FullAdd4CL))).
connected(fadd4_cl,out(u5(FullAdd4CL)),in1(u6(FullAdd4CL))).
connected(fadd4_cl,out(u4(FullAdd4CL)),in0(u6(FullAdd4CL))).
connected(fadd4_cl,carryin(FullAdd4CL),in0(u22(FullAdd4CL))).
connected(fadd4_cl,sum(u0(FullAdd4CL)),in1(u22(FullAdd4CL))).

connected(fadd4_cl,sum(u1(FullAdd4CL)),in1(u7(FullAdd4CL))).
connected(fadd4_cl,carry(u0(FullAdd4CL)),in0(u7(FullAdd4CL))).
connected(fadd4_cl,out(u6(FullAdd4CL)),in0(u23(FullAdd4CL))).
connected(fadd4_cl,sum(u1(FullAdd4CL)),in1(u23(FullAdd4CL))).
connected(fadd4_cl,sum(u1(FullAdd4CL)),in0(u8(FullAdd4CL))).
connected(fadd4_cl,carryin(FullAdd4CL),in1(u8(FullAdd4CL))).
connected(fadd4_cl,sum(u0(FullAdd4CL)),in2(u8(FullAdd4CL))).
connected(fadd4_cl,carry(u1(FullAdd4CL)),in(u9(FullAdd4CL))).
connected(fadd4_cl,out(u7(FullAdd4CL)),in0(u10(FullAdd4CL))).
connected(fadd4_cl,out(u8(FullAdd4CL)),in1(u10(FullAdd4CL))).
connected(fadd4_cl,out(u9(FullAdd4CL)),in3(u10(FullAdd4CL))).

connected(fadd4_cl,sum(u2(FullAdd4CL)),in1(u24(FullAdd4CL))).
connected(fadd4_cl,out(u10(FullAdd4CL)),in0(u24(FullAdd4CL))).
connected(fadd4_cl,carry(u1(FullAdd4CL)),in0(u11(FullAdd4CL))).
connected(fadd4_cl,sum(u2(FullAdd4CL)),in1(u11(FullAdd4CL))).
connected(fadd4_cl,sum(u2(FullAdd4CL)),in0(u12(FullAdd4CL))).
connected(fadd4_cl,carry(u0(FullAdd4CL)),in1(u12(FullAdd4CL))).
connected(fadd4_cl,sum(u1(FullAdd4CL)),in2(u12(FullAdd4CL))).
connected(fadd4_cl,sum(u2(FullAdd4CL)),in0(u13(FullAdd4CL))).
connected(fadd4_cl,sum(u1(FullAdd4CL)),in1(u13(FullAdd4CL))).
connected(fadd4_cl,carryin(FullAdd4CL),in2(u13(FullAdd4CL))).
connected(fadd4_cl,sum(u0(FullAdd4CL)),in3(u13(FullAdd4CL))).
connected(fadd4_cl,carry(u2(FullAdd4CL)),in(u14(FullAdd4CL))).
connected(fadd4_cl,out(u11(FullAdd4CL)),in0(u15(FullAdd4CL))).
connected(fadd4_cl,out(u12(FullAdd4CL)),in1(u15(FullAdd4CL))).
connected(fadd4_cl,out(u13(FullAdd4CL)),in2(u15(FullAdd4CL))).
connected(fadd4_cl,out(u14(FullAdd4CL)),in3(u15(FullAdd4CL))).

connected(fadd4_cl,sum(u3(FullAdd4CL)),in1(u25(FullAdd4CL))).
connected(fadd4_cl,out(u15(FullAdd4CL)),in0(u25(FullAdd4CL))).
connected(fadd4_cl,carry(u2(FullAdd4CL)),in0(u16(FullAdd4CL))).
connected(fadd4_cl,sum(u3(FullAdd4CL)),in1(u16(FullAdd4CL))).
connected(fadd4_cl,sum(u2(FullAdd4CL)),in0(u17(FullAdd4CL))).
connected(fadd4_cl,sum(u3(FullAdd4CL)),in1(u17(FullAdd4CL))).
connected(fadd4_cl,carry(u1(FullAdd4CL)),in2(u17(FullAdd4CL))).
connected(fadd4_cl,carry(u0(FullAdd4CL)),in0(u18(FullAdd4CL))).
connected(fadd4_cl,sum(u3(FullAdd4CL)),in1(u18(FullAdd4CL))).
```

```
connected(fadd4_cl,sum(u2(FullAdd4CL)),in2(u18(FullAdd4CL))).
connected(fadd4_cl,sum(u1(FullAdd4CL)),in3(u18(FullAdd4CL))).
connected(fadd4_cl,sum(u3(FullAdd4CL)),in0(u19(FullAdd4CL))).
connected(fadd4_cl,sum(u2(FullAdd4CL)),in1(u19(FullAdd4CL))).
connected(fadd4_cl,sum(u1(FullAdd4CL)),in2(u19(FullAdd4CL))).
connected(fadd4_cl,carryin(FullAdd4CL),in3(u19(FullAdd4CL))).
connected(fadd4_cl,sum(u0(FullAdd4CL)),in4(u19(FullAdd4CL))).
connected(fadd4_cl,carry(u3(FullAdd4CL)),in(u20(FullAdd4CL))).
connected(fadd4_cl,out(u16(FullAdd4CL)),in0(u21(FullAdd4CL))).
connected(fadd4_cl,out(u17(FullAdd4CL)),in1(u21(FullAdd4CL))).
connected(fadd4_cl,out(u18(FullAdd4CL)),in2(u21(FullAdd4CL))).
connected(fadd4_cl,out(u19(FullAdd4CL)),in3(u21(FullAdd4CL))).
connected(fadd4_cl,out(u20(FullAdd4CL)),in4(u21(FullAdd4CL))).

connected(fadd4_cl,out(u21(FullAdd4CL)),carryout(FullAdd4CL)).
                                        % u21 produced carryout

connected(fadd4_cl,out(u22(FullAdd4CL)),sum0(FullAdd4CL)).
                                        % u22 produces sum0
connected(fadd4_cl,out(u23(FullAdd4CL)),sum1(FullAdd4CL)).
                                        % u23 produces sum1
connected(fadd4_cl,out(u24(FullAdd4CL)),sum2(FullAdd4CL)).
                                        % u24 produces sum2
connected(fadd4_cl,out(u25(FullAdd4CL)),sum3(FullAdd4CL)).
                                        % u25 produces sum3


/* Behavioral Specification for an full adder */

output_eqn(fadd4_cl,sum0(FA4CL) :=
                    or(
                       and( carryin(FA4CL),
                           or( and( in00(FA4CL),in10(FA4CL)),
                               and( neg( in00(FA4CL)),neg( in10(FA4CL))))),
                       and( neg( carryin(FA4CL)),
                           or( and( in00(FA4CL),neg( in10(FA4CL))),
                               and( neg( in00(FA4CL)),in10(FA4CL))))) ).

output_eqn(fadd4_cl,sum1(FA4CL) :=
       or( and( neg( or( and( in01(FA4CL),neg( in11(FA4CL))),
                         and( neg( in01(FA4CL)),in11(FA4CL)) ) ),
                or( and( in00(FA4CL),in10(FA4CL)),
                    and( carryin(FA4CL),
                         or( and( in00(FA4CL),neg( in10(FA4CL))),
                             and( neg( in00(FA4CL)),in10(FA4CL)) ))) ),
```

```
            and( or( and( in01(FA4CL),neg( in11(FA4CL))),
                     and( neg( in01(FA4CL)),in11(FA4CL)) ),
                 neg( or( and( in00(FA4CL),in10(FA4CL)),
                          and( carryin(FA4CL),
                               or( and( in00(FA4CL),
                                        neg( in10(FA4CL))),
                                   and( neg( in00(FA4CL)),
                                        in10(FA4CL)))))))).

output_eqn(fadd4_cl,sum2(FA4CL) :=
       or( and( neg( or( and(in02(FA4CL),neg( in12(FA4CL))),
                         and( neg( in02(FA4CL)),in12(FA4CL)) )),
                or( and( in01(FA4CL),in11(FA4CL)),
                    and( or( in01(FA4CL),in11(FA4CL)),
                         or( or( and( carryin(FA4CL),in00(FA4CL)),
                                 and( carryin(FA4CL),in10(FA4CL))),
                         and( in00(FA4CL),in10(FA4CL)))))),
           and( or( and(in02(FA4CL),neg( in12(FA4CL))),
                    and( neg( in02(FA4CL)),in12(FA4CL)) ),
                neg( or( and( in01(FA4CL),in11(FA4CL)),
                         and( or( in01(FA4CL),in11(FA4CL)),
                              or( or( and( carryin(FA4CL),in00(FA4CL)),
                                      and( carryin(FA4CL),in10(FA4CL))),
                              and( in00(FA4CL),in10(FA4CL))))) ))) ).

output_eqn(fadd4_cl,sum3(FA4CL) :=
   or( and( neg( or( and(in03(FA4CL),neg( in13(FA4CL))),
                     and( neg( in03(FA4CL)),in13(FA4CL)) )),
            or( and( in02(FA4CL),in12(FA4CL)),
                and( or( in02(FA4CL),in12(FA4CL)),
                     or( or( and( or( and( carryin(FA4CL),
                                           or( and( in00(FA4CL),
                                                    neg( in10(FA4CL))),
                                               and( neg( in00(FA4CL)),
                                                    in10(FA4CL)))),
                                      and( in00(FA4CL),in10(FA4CL))),
                                 in11(FA4CL)),
                             and( in01(FA4CL),in11(FA4CL))),
                         and( or( and( carryin(FA4CL),
                                       or( and( in00(FA4CL),
                                                neg( in10(FA4CL))),
                                           and( neg( in00(FA4CL)),
                                                in10(F)))),
                              and( in00(FA4CL),in10(FA4CL))),
                         in01(FA4CL)))))),
```

```
        and( or( and(in03(FA4CL),neg( in13(FA4CL))),
                 and( neg( in03(FA4CL)),in13(FA4CL)) ),
             neg( or( and( in02(FA4CL),in12(FA4CL)),
                      and( or( in02(FA4CL),in12(FA4CL)),
                           or( or( and( or( and( carryin(FA4CL),
                                                 or( and( in00(FA4CL),
                                                          neg( in10(FA4CL))),
                                                     and( neg( in00(FA4CL)),
                                                          in10(F)))),
                                               and( in00(FA4CL),in10(FA4CL))),
                                           in11(FA4CL)),
                                       and( in01(FA4CL),in11(FA4CL))),
                                   and( or( and( carryin(FA4CL),
                                                 or( and( in00(FA4CL),
                                                          neg( in10(FA4CL))),
                                                     and( neg( in00(FA4CL)),
                                                          in10(F)))),
                                               and( in00(FA4CL),in10(FA4CL))),
                                           in01(FA4CL)))))) ))).


output_eqn(fadd4_cl,carryout(FA4CL) :=
            or( and( in03(FA4CL),in13(FA4CL)),
                and( or( in03(FA4CL),in13(FA4CL)),
                     or( or( and( or( and( in01(FA4CL),in11(FA4CL)),
                                      and( or( in01(FA4CL),in11(FA4CL)),
                                           or( or( and( carryin(FA4CL),
                                                        in00(FA4CL)),
                                                   and( carryin(FA4CL),
                                                        in10(FA4CL))),
                                               and( in00(FA4CL),in10(FA4CL))))),
                                  in02(FA4CL)),
                              and( or( and( in01(FA4CL),in11(FA4CL)),
                                       and( or( in01(FA4CL),in11(FA4CL)),
                                            or( or( and( carryin(FA4CL),
                                                         in00(FA4CL)),
                                                    and( carryin(FA4CL),
                                                         in10(FA4CL))),
                                                and( in00(FA4CL),
                                                     in10(FA4CL))))),
                                   in12(FA4CL)),
                          and( in02(FA4CL),in22(FA4CL)))))).
```

```
/*****************************************************************
--- Adapted from Zycad VHDL File:
-----------------------------------------------------------------
-- DATE: 17 April 1991
-- VERSION:  1
-- UNIX FILENAME:  four_bit_cl_adder_entity.vhd
-- FUNCTION: This file contains the entity and structural
-- architecture for a four bit carry look ahead adder.
-- AUTHOR: dwb (Capt David W Banton)
-----------------------------------------------------------------
library ZYCAD;
use     ZYCAD.types.all;
use     ZYCAD.COMPONENTS.all;
--
--  THE ENTITY DECLARATION:
--
entity four_bit_cl_adder is                        -- 4b carry look-ahead adder
  port(Input0,                                     -- 4 bit word input 0
       Input1:   In  MVL7_Vector (3 downto 0); -- 4 bit word input 1
       CarryIn:  In  MVL7;                       -- carry input
       Sum:      Out MVL7_Vector (3 downto 0); -- 4 bit word output
       CarryOut: Out MVL7);                      -- carry output
end four_bit_cl_adder;
--
-- THE ARCHITECTURAL BODY:
--
architecture Structural of four_bit_cl_adder is
--
signal P0, P1, P2, P3, G0, G1, G2, G3, G0not, G1not, G2not, G3not,
       S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, C0, C1, C2: MVL7;
--
component invgate                              -- inverter
  generic (tLH: Time;                          -- rise inertial delay
           tHL: Time);                         -- fall inertial delay
  port (Input: In MVL7;                        -- input
        Output: Out MVL7);                     -- output
end component;
--
component nandgate                             -- N input NAND gate
  generic (N: Positive;                        -- number of inputs
           tLH: Time;                          -- rise inertial delay
           tHL: Time);                         -- fall inertial delay
  port (Input:  In  MVL7_Vector (1 to N); -- input
        Output: Out MVL7);                     -- output
```

```
end component;
--
component xorgate
  generic (N: Positive;                    -- number of inputs
           tLH: Time;                      -- rise inertial delay
           tHL: Time);                     -- fall inertial delay
  port (Input:  In  MVL7_Vector (1 to N); -- input
        Output: Out MVL7);                -- output
end component;
--
component half_adder                       -- half_adder
  port (Input: In  MVL7_Vector (1 to 2);  -- input
        Sum,                               -- sum
        Carry: Out MVL7);                  -- carry
end component;
--
begin
  --  Half_adders:
  U0  : half_adder port map (Input(1) => Input0(0), Input(2) => Input1(0),
                             Sum      => P0,        Carry    => G0);
  U1  : half_adder port map (Input(1) => Input0(1), Input(2) => Input1(1),
                             Sum      => P1,        Carry    => G1);
  U2  : half_adder port map (Input(1) => Input0(2), Input(2) => Input1(2),
                             Sum      => P2,        Carry    => G2);
  U3  : half_adder port map (Input(1) => Input0(3), Input(2) => Input1(3),
                             Sum      => P3,        Carry    => G3);
  --  Sum and Carry 0:
  U4  : nandgate generic map (2, 2 ns, 2 ns)
                port map (Input(1) => CarryIn,   Input(2) => P0,
                          Output   => S0);
  U5  : invgate  generic map (1 ns, 1 ns)
                port map (G0, G0not);
  U6  : nandgate generic map (2, 2 ns, 2 ns)
                port map (Input(1) => G0not,     Input(2) => S0,
                          Output   => C0);
  --  Sum and Carry 1:
  U7  : nandgate generic map (2, 2 ns, 2 ns)
                port map (Input(1) => G0,        Input(2) => P1,
                          Output   => S1);
  U8  : nandgate generic map (3, 3 ns, 3 ns)
                port map (Input(1) => CarryIn,   Input(2) => P0,
                          Input(3) => P1,        Output   => S2);
  U9  : invgate  generic map (1 ns, 1 ns)
                port map (G1, G1not);
  U10 : nandgate generic map (3, 3 ns, 3 ns)
```

```
                port map (Input(1) => G1not,     Input(2) => S1,
                          Input(3) => S2,        Output   => C1);
  --  Sum and Carry 2:
  U11 : nandgate generic map (2, 2 ns, 2 ns)
                port map (Input(1) => G1,        Input(2) => P2,
                          Output   => S3);
  U12 : nandgate generic map (3, 3 ns, 3 ns)
                port map (Input(1) => G0,        Input(2) => P1,
                          Input(3) => P2,        Output   => S4);
  U13 : nandgate generic map (4, 4 ns, 4 ns)
                port map (Input(1) => CarryIn,   Input(2) => P0,
                          Input(3) => P1,        Input(4) => P2,
                          Output   => S5);
  U14 : invgate  generic map (1 ns, 1 ns)
                port map (G2, G2not);
  U15 : nandgate generic map (4, 4 ns, 4 ns)
                port map (Input(1) => G2not,     Input(2) => S3,
                          Input(3) => S4,        Input(4) => S5,
                          Output   => C2);
  --  Sum and Carry 3:
  U16 : nandgate generic map (2, 2 ns, 2 ns)
                port map (Input(1) => G2,        Input(2) => P3,
                          Output   => S6);
  U17 : nandgate generic map (3, 3 ns, 3 ns)
                port map (Input(1) => G1,        Input(2) => P2,
                          Input(3) => P3,        Output   => S7);
  U18 : nandgate generic map (4, 4 ns, 4 ns)
                port map (Input(1) => G0,        Input(2) => P1 ,
                          Input(3) => P2,        Input(4) => P3,
                          Output   => S8);
  U19 : nandgate generic map (5, 5 ns, 5 ns)
                port map (Input(1) => CarryIn,   Input(2) => P0,
                          Input(3) => P1,        Input(4) => P2,
                          Input(5) => P3,        Output   => S9);
  U20 : invgate  generic map (1 ns, 1 ns)
                port map (G3, G3not);
  U21 : nandgate generic map (5, 5 ns, 5 ns)
                port map (Input(1) => G3not,     Input(2) => S6,
                          Input(3) => S7,        Input(4) => S8,
                          Input(5) => S9,        Output   => CarryOut);
--
  U22 : xorgate  generic map (2, 2 ns, 2 ns)
                port map (Input(1) => P0,        Input(2) => CarryIn,
                          Output   => Sum(0));
  U23 : xorgate  generic map (2, 2 ns, 2 ns)
```

```
                port map (Input(1) => P1,        Input(2) => C0,
                          Output   => Sum(1));
  U24 : xorgate generic map (2, 2 ns, 2 ns)
                port map (Input(1) => P2,        Input(2) => C1,
                          Output   => Sum(2));
  U25 : xorgate generic map (2, 2 ns, 2 ns)
                port map (Input(1) => P3,        Input(2) => C2,
                          Output   => Sum(3));
end Structural;

**********************************************************************/
```

Appendix B. *Sample Program Runs*

B.1 Sample Verification Run Using Sparks's **AFIT_VERIFY**

B.1.1 Verification of One-Bit Full Adder faddxor.pl

```
Script started on Fri Aug  2 14:30:47 1991
csh> prolog

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- ['qfaddld.pro'].
[consulting /usr/users/ela/labovitz/newverify/qfaddld.pro...]
 [consulting /usr/users/ela/labovitz/newverify/qops.pro...]
[Undefined procedures will just fail ('fail' option)]
 [qops.pro consulted 0.267 sec 1,092 bytes]
 [consulting /usr/users/ela/labovitz/newverify/eval.pro...]
[WARNING: Singleton variables, clause 3 of evaluate_brown/2: F]
 [eval.pro consulted 0.700 sec 2,936 bytes]
 [consulting /usr/users/ela/labovitz/newverify/derbeh.pro...]
[WARNING: Singleton variables, clause 1 of derive_behaviors/3: Spec_Behavior]
[WARNING: Singleton variables, clause 2 of derive_behaviors/3: Spec_Behavior]
[WARNING: Clauses for derive_behavior/3 are not together in the source file]
[WARNING: Singleton variables, clause 1 of derive_behavior/3: F]
[WARNING: Singleton variables, clause 2 of derive_behavior/3: F]
[WARNING: Singleton variables, clause 8 of derive_behavior/3: Module]
 [derbeh.pro consulted 0.984 sec 3,220 bytes]
 [consulting /usr/users/ela/labovitz/newverify/derstate.pro...]
[WARNING: Singleton variables, clause 1 of derive_states/3: Type, Part]
[WARNING: Singleton variables, clause 1 of replace_all/5: Part]
[WARNING: Singleton variables, clause 2 of replace_all/5: Module, Old, New]
[WARNING: Singleton variables, clause 1 of replace/4: Old, New]
[WARNING: Singleton variables, clause 2 of replace/4: Old, New]
[WARNING: Singleton variables, clause 3 of replace/4: New, Arg1]
[WARNING: Singleton variables, clause 11 of replace/4: Old, New]
 [derstate.pro consulted 0.966 sec 2,904 bytes]
 [consulting /usr/users/ela/labovitz/newverify/xor.pro...]
[WARNING: Singleton variables, clause 1 of port/4: ANand2]
[WARNING: Singleton variables, clause 2 of port/4: ANand2]
[WARNING: Singleton variables, clause 3 of port/4: ANand2]
[WARNING: Clauses for module_name/1 are not together in the source file]
[WARNING: Clauses for port/4 are not together in the source file]
[WARNING: Singleton variables, clause 1 of port/4: AnXor]
```

```
[WARNING: Singleton variables, clause 2 of port/4: AnXor]
[WARNING: Singleton variables, clause 3 of port/4: AnXor]
[WARNING: Singleton variables, clause 1 of part/3: AnXor]
[WARNING: Singleton variables, clause 2 of part/3: AnXor]
[WARNING: Singleton variables, clause 3 of part/3: AnXor]
[WARNING: Singleton variables, clause 4 of part/3: AnXor]
[WARNING: Clauses for output_eqn/2 are not together in the source file]
 [xor.pro consulted 0.816 sec 2,616 bytes]
 [consulting /usr/users/ela/labovitz/newverify/faddxcr.pro...]
[WARNING: Singleton variables, clause 1 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 2 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 3 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 4 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 5 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 1 of part/3: Afaddxor]
[WARNING: Singleton variables, clause 2 of part/3: Afaddxor]
[WARNING: Singleton variables, clause 3 of part/3: Afaddxor]
[WARNING: Singleton variables, clause 4 of part/3: Afaddxor]
[WARNING: Singleton variables, clause 5 of part/3: Afaddxor]
 [faddxor.pro consulted 0.766 sec 2,368 bytes]
 [consulting /usr/users/ela/labovitz/newverify/boole2.pro...]
[WARNING: Singleton variables, clause 1 of remove_x_1/3: X]
[WARNING: Singleton variables, clause 2 of remove_x_1/3: X]
[WARNING: Singleton variables, clause 3 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 4 of remove_x_1/3: Arg, Arg1, Arg2]
[WARNING: Singleton variables, clause 5 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 6 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 1 of remove_x_0/3: X]
[WARNING: Singleton variables, clause 2 of remove_x_0/3: X]
[WARNING: Singleton variables, clause 3 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 4 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 5 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 6 of remove_x_0/3: Arg2]
 [boole2.pro consulted 1.750 sec 6,900 bytes]
 [consulting /usr/users/ela/labovitz/newverify/eqbeh.pro...]
[WARNING: Singleton variables, clause 1 of eqb/3: M]
[WARNING: Singleton variables, clause 2 of eqb/3: M]
[WARNING: Clauses for eqb/3 are not together in the source file]
 [eqbeh.pro consulted 0.400 sec 1,236 bytes]
 [consulting /usr/users/ela/labovitz/newverify/verify.pro...]
 [verify.pro consulted 0.800 sec 3,180 bytes]
[qfaddld.pro consulted 8.117 sec 27,260 bytes]

yes
| ?- verify(faddxor).
```

```
>>> Attempting to verify faddxor>>>
>>>nand2 primitive (needs no verificatien)>>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
>>> Attempting to verify xor>>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
component list is [nand2]
Applying Rule 1B to out(_1068)
Applying Rule 2A to out(g4(_1068))
nand2's output equation:
   out(g4(_1068)) := or(neg(in0(g4(_1068))),neg(in1(g4(_1068))))
Applying Rule 5 to or(neg(in0(g4(_1068))),neg(in1(g4(_1068))))
Applying Rule 3 to neg(in0(g4(_1068)))
Applying Rule 1B to in0(g4(_1068))
Applying Rule 2A to out(g2(_1068))
nand2's output equation:
   out(g2(_1068)) := or(neg(in0(g2(_1068))),neg(in1(g2(_1068))))
Applying Rule 5 to or(neg(in0(g2(_1068))),neg(in1(g2(_1068))))
Applying Rule 3 to neg(in0(g2(_1068)))
Applying Rule 1A to in0(g2(_1068))
Value of neg(in0(_1068)):
          neg(in0(_1068))
Applying Rule 3 to neg(in1(g2(_1068)))
Applying Rule 1B to in1(g2(_1068))
Applying Rule 2A to out(g1(_1068))
nand2's output equation:
   out(g1(_1068)) := or(neg(in0(g1(_1068))),neg(in1(g1(_1068))))
Applying Rule 5 to or(neg(in0(g1(_1068))),neg(in1(g1(_1068))))
Applying Rule 3 to neg(in0(g1(_1068)))
Applying Rule 1A to in0(g1(_1068))
Value of neg(in0(_1068)):
          neg(in0(_1068))
Applying Rule 3 to neg(in1(g1(_1068)))
Applying Rule 1A to in1(g1(_1068))
Value of neg(in1(_1068)):
          neg(in1(_1068))
Value of or(neg(in0(_1068)),neg(in1(_1068))):
          or(neg(in0(_1068)),neg(in1(_1068)))
Value of neg(or(neg(in0(_1068)),neg(in1(_1068)))):
          and(in0(_1068),in1(_1068))
Value of or(neg(in0(_1068)),and(in0(_1068),in1(_1068))):
          or(neg(in0(_1068)),and(in0(_1068),in1(_1068)))
```

```
Value of neg(or(neg(in0(_1068)),and(in0(_1068),in1(_1068)))):
          and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068))))
Applying Rule 3 to neg(in1(g4(_1068)))
Applying Rule 1B to in1(g4(_1068))
Applying Rule 2A to out(g3(_1068))
nand2's output equation:
   out(g3(_1068)) := or(neg(in0(g3(_1068))),neg(in1(g3(_1068))))
Applying Rule 5 to or(neg(in0(g3(_1068))),neg(in1(g3(_1068))))
Applying Rule 3 to neg(in0(g3(_1068)))
Applying Rule 1B to in0(g3(_1068))
Applying Rule 2A to out(g1(_1068))
nand2's output equation:
   out(g1(_1068)) := or(neg(in0(g1(_1068))),neg(in1(g1(_1068))))
Applying Rule 5 to or(neg(in0(g1(_1068))),neg(in1(g1(_1068))))
Applying Rule 3 to neg(in0(g1(_1068)))
Applying Rule 1A to in0(g1(_1068))
Value of neg(in0(_1068)):
          neg(in0(_1068))
Applying Rule 3 to neg(in1(g1(_1068)))
Applying Rule 1A to in1(g1(_1068))
Value of neg(in1(_1068)):
          neg(in1(_1068))
Value of or(neg(in0(_1068)),neg(in1(_1068))):
          or(neg(in0(_1068)),neg(in1(_1068)))
Value of neg(or(neg(in0(_1068)),neg(in1(_1068)))):
          and(in0(_1068),in1(_1068))
Applying Rule 3 to neg(in1(g3(_1068)))
Applying Rule 1A to in1(g3(_1068))
Value of neg(in1(_1068)):
          neg(in1(_1068))
Value of or(and(in0(_1068),in1(_1068)),neg(in1(_1068))):
          or(and(in0(_1068),in1(_1068)),neg(in1(_1068)))
Value of neg(or(and(in0(_1068),in1(_1068)),neg(in1(_1068)))):
          and(or(neg(in0(_1068)),neg(in1(_1068))),in1(_1068))
Value of or(and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068))),
                           and(or(neg(in0(_1068)),neg(in1(_1068))),
                               in1(_1068))):
          or(and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068)))),
             and(or(neg(in0(_1068)),neg(in1(_1068))),in1(_1068)))
Does or(and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068)))),
        and(or(neg(in0(_1068)),neg(in1(_1068))),in1(_1068))) =
      or(and(neg(in0(_1068)),in1(_1068)),and(in0(_1068),
                                             neg(in1(_1068))))
or(and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068)))),
       and(or(neg(in0(_1068)),neg(in1(_1068))),in1(_1068))) =
```

```
     or(and(neg(in0(_1068)),in1(_1068)),and(in0(_1068),neg(in1(_1068))))
By Boolean Expansion
output list is [out(_1087)]
derived list is[out(_1159)]
Outnum is1
Derived  number is1
<<< Success! Behavior of xor meets its specification.
>>>xor previously verified >>>
component list is [nand2]
Applying Rule 1B to outcarry(_804)
Applying Rule 2A to out(g3(_804))
nand2's output equation:
   out(g3(_804)) := or(neg(in0(g3(_804))),neg(in1(g3(_804))))
Applying Rule 5 to or(neg(in0(g3(_804))),neg(in1(g3(_804))))
Applying Rule 3 to neg(in0(g3(_804)))
Applying Rule 1B to in0(g3(_804))
Applying Rule 2A to out(g1(_804))
nand2's output equation:
   out(g1(_804)) := or(neg(in0(g1(_804))),neg(in1(g1(_804))))
Applying Rule 5 to or(neg(in0(g1(_804))),neg(in1(g1(_804))))
Applying Rule 3 to neg(in0(g1(_804)))
Applying Rule 1A to in0(g1(_804))
Value of neg(x(_804)):
          neg(x(_804))
Applying Rule 3 to neg(in1(g1(_804)))
Applying Rule 1A to in1(g1(_804))
Value of neg(y(_804)):
          neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
          or(neg(x(_804)),neg(y(_804)))
Value of neg(or(neg(x(_804)),neg(y(_804)))):
          and(x(_804),y(_804))
Applying Rule 3 to neg(in1(g3(_804)))
Applying Rule 1B to in1(g3(_804))
Applying Rule 2A to out(g2(_804))
nand2's output equation:
   out(g2(_804)) := or(neg(in0(g2(_804))),neg(in1(g2(_804))))
Applying Rule 5 to or(neg(in0(g2(_804))),neg(in1(g2(_804))))
Applying Rule 3 to neg(in0(g2(_804)))
Applying Rule 1A to in0(g2(_804))
Value of neg(cin(_804)):
          neg(cin(_804))
Applying Rule 3 to neg(in1(g2(_804)))
Applying Rule 1B to in1(g2(_804))
Applying Rule 2B to out(g4(_804))
```

```
xor's derived behavior:
   out(g4(_804)) := or(and(in0(g4(_804)),or(neg(in0(g4(_804))),
                    neg(in1(g4(_804))))),and(or(neg(in0(g4(_804))),
                    neg(in1(g4(_804)))),in1(g4(_804))))
Applying Rule 5 to or(and(in0(g4(_804)),or(neg(in0(g4(_804))),
                     neg(in1(g4(_804))))),and(or(neg(in0(g4(_804))),
                     neg(in1(g4(_804)))),in1(g4(_804))))
Applying Rule 4 to and(in0(g4(_804)),or(neg(in0(g4(_804))),
                     neg(in1(g4(_804)))))
Applying Rule 1A to in0(g4(_804))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
          neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
          neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
          or(neg(x(_804)),neg(y(_804)))
Value of and(x(_804),or(neg(x(_804)),neg(y(_804)))):
          and(x(_804),or(neg(x(_804)),neg(y(_804))))
Applying Rule 4 to and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                      in1(g4(_804)))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
          neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
          neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
          or(neg(x(_804)),neg(y(_804)))
Applying Rule 1A to in1(g4(_804))
Value of and(or(neg(x(_804)),neg(y(_804))),y(_804)):
          and(or(neg(x(_804)),neg(y(_804))),y(_804))
Value of or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),and(or(neg(x(_804)),
                neg(y(_804))),y(_804))):
          or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),and(or(neg(x(_804)),
                neg(y(_804))),y(_804)))
Value of neg(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                and(or(neg(x(_804)),neg(y(_804))),y(_804)))):
```

```
          and(or(neg(x(_804)),and(x(_804),y(_804))),or(and(x(_804),
               y(_804)),neg(y(_804))))
Value of or(neg(cin(_804)),and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804))))):
          or(neg(cin(_804)),and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))))
Value of neg(or(neg(cin(_804)),and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804))))))):
          and(cin(_804),or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))))
Value of or(and(x(_804),y(_804)),and(cin(_804),or(and(x(_804),
               or(neg(x(_804)),neg(y(_804)))),and(or(neg(x(_804)),
               neg(y(_804))),y(_804))))):
          or(and(x(_804),y(_804)),and(cin(_804),or(and(x(_804),
               or(neg(x(_804)),neg(y(_804)))),and(or(neg(x(_804)),
               neg(y(_804))),y(_804)))))
Does or(and(x(_804),y(_804)),and(cin(_804),or(and(x(_804),
               or(neg(x(_804)),neg(y(_804)))),and(or(neg(x(_804)),
               neg(y(_804))),y(_804)))))) =
      or(and(x(_804),y(_804)),and(cin(_804),xor(x(_804),y(_804))))
or(and(x(_804),y(_804)),and(cin(_804),or(and(x(_804),
               or(neg(x(_804)),neg(y(_804)))),and(or(neg(x(_804)),
               neg(y(_804))),y(_804)))))) =
     or(and(x(_804),y(_804)),and(cin(_804),xor(x(_804),y(_804))))
By Boolean Expansion
Applying Rule 1B to outsum(_804)
Applying Rule 2B to out(g5(_804))
xor's derived behavior:
   out(g5(_804)) := or(and(in0(g5(_804)),or(neg(in0(g5(_804))),
               neg(in1(g5(_804))))),and(or(neg(in0(g5(_804))),
               neg(in1(g5(_804)))),in1(g5(_804))))
Applying Rule 5 to or(and(in0(g5(_804)),or(neg(in0(g5(_804))),
               neg(in1(g5(_804))))),and(or(neg(in0(g5(_804))),
               neg(in1(g5(_804)))),in1(g5(_804))))
Applying Rule 4 to and(in0(g5(_804)),or(neg(in0(g5(_804))),
               neg(in1(g5(_804)))))
Applying Rule 1B to in0(g5(_804))
Applying Rule 2B to out(g4(_804))
xor's derived behavior:
   out(g4(_804)) := or(and(in0(g4(_804)),or(neg(in0(g4(_804))),
               neg(in1(g4(_804))))),and(or(neg(in0(g4(_804))),
               neg(in1(g4(_804)))),in1(g4(_804))))
Applying Rule 5 to or(and(in0(g4(_804)),or(neg(in0(g4(_804))),
               neg(in1(g4(_804))))),and(or(neg(in0(g4(_804))),
               neg(in1(g4(_804)))),in1(g4(_804))))
```

```
Applying Rule 4 to and(in0(g4(_804)),or(neg(in0(g4(_804))),
               neg(in1(g4(_804)))))
Applying Rule 1A to in0(g4(_804))
Ar 'ying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Value of and(x(_804),or(neg(x(_804)),neg(y(_804)))):
         and(x(_804),or(neg(x(_804)),neg(y(_804))))
Applying Rule 4 to and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
         in1(g4(_804)))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Applying Rule 1A to in1(g4(_804))
Value of and(or(neg(x(_804)),neg(y(_804))),y(_804)):
         and(or(neg(x(_804)),neg(y(_804))),y(_804))
Value of or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))):
         or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804)))
Applying Rule 5 to or(neg(in0(g5(_804))),neg(in1(g5(_804))))
Applying Rule 3 to neg(in0(g5(_804)))
Applying Rule 1B to in0(g5(_804))
Applying Rule 2B to out(g4(_804))
xor's derived behavior:
   out(g4(_804)) := or(and(in0(g4(_804)),or(neg(in0(g4(_804))),
         neg(in1(g4(_804))))),and(or(neg(in0(g4(_804))),
         neg(in1(g4(_804)))),in1(g4(_804))))
Applying Rule 5 to or(and(in0(g4(_804)),or(neg(in0(g4(_804))),
```

```
          neg(in1(g4(_804)))))),and(or(neg(in0(g4(_804))),
          neg(in1(g4(_804)))),in1(g4(_804))))
Applying Rule 4 to and(in0(g4(_804)),or(neg(in0(g4(_804))),
          neg(in1(g4(_804)))))
Applying Rule 1A to in0(g4(_804))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
          neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
          neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
          or(neg(x(_804)),neg(y(_804)))
Value of and(x(_804),or(neg(x(_804)),neg(y(_804)))):
          and(x(_804),or(neg(x(_804)),neg(y(_804))))
Applying Rule 4 to and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
          in1(g4(_804)))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
          neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
          neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
          or(neg(x(_804)),neg(y(_804)))
Applying Rule 1A to in1(g4(_804))
Value of and(or(neg(x(_804)),neg(y(_804))),y(_804)):
          and(or(neg(x(_804)),neg(y(_804))),y(_804))
Value of or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))):
          or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804)))
Value of neg(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804)))):
          and(or(neg(x(_804)),and(x(_804),y(_804))),or(and(x(_804),y(_804)),
               neg(y(_804))))
Applying Rule 3 to neg(in1(g5(_804)))
Applying Rule 1A to in1(g5(_804))
Value of neg(cin(_804)):
```

```
          neg(cin(_804))
Value of or(and(or(neg(x(_804)),and(x(_804),y(_804))),or(and(x(_804),
               y(_804)),neg(y(_804)))),neg(cin(_804))):
          or(and(or(neg(x(_804)),and(x(_804),y(_804))),or(and(x(_804),
               y(_804)),neg(y(_804)))),neg(cin(_804)))
Value of and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
          and(or(neg(x(_804)),neg(y(_804))),y(_804))),
               or(and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))),neg(cin(_804)))):
          and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))),
               or(and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))),neg(cin(_804))))
Applying Rule 4 to and(or(neg(in0(g5(_804))),neg(in1(g5(_804)))),
          in1(g5(_804)))
Applying Rule 5 to or(neg(in0(g5(_804))),neg(in1(g5(_804))))
Applying Rule 3 to neg(in0(g5(_804)))
Applying Rule 1B to in0(g5(_804))
Applying Rule 2B to out(g4(_804))
xor's derived behavior:
   out(g4(_804)) := or(and(in0(g4(_804)),or(neg(in0(g4(_804))),
          neg(in1(g4(_804))))),and(or(neg(in0(g4(_804))),
          neg(in1(g4(_804)))),in1(g4(_804))))
Applying Rule 5 to or(and(in0(g4(_804)),or(neg(in0(g4(_804))),
          neg(in1(g4(_804))))),and(or(neg(in0(g4(_804))),
          neg(in1(g4(_804)))),in1(g4(_804))))
Applying Rule 4 to and(in0(g4(_804)),or(neg(in0(g4(_804))),
          neg(in1(g4(_804)))))
Applying Rule 1A to in0(g4(_804))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
          neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
          neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
          or(neg(x(_804)),neg(y(_804)))
Value of and(x(_804),or(neg(x(_804)),neg(y(_804)))):
          and(x(_804),or(neg(x(_804)),neg(y(_804))))
Applying Rule 4 to and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
          in1(g4(_804)))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
```

```
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Applying Rule 1A to in1(g4(_804))
Value of and(or(neg(x(_804)),neg(y(_804))),y(_804)):
         and(or(neg(x(_804)),neg(y(_804))),y(_804))
Value of or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))):
         or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804)))
Value of neg(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804)))):
         and(or(neg(x(_804)),and(x(_804),y(_804))),or(and(x(_804),
               y(_804)),neg(y(_804))))
Applying Rule 3 to neg(in1(g5(_804)))
Applying Rule 1A to in1(g5(_804))
Value of neg(cin(_804)):
         neg(cin(_804))
Value of or(and(or(neg(x(_804)),and(x(_804),y(_804))),
              or(and(x(_804),y(_804)),neg(y(_804)))),neg(cin(_804))):
         or(and(or(neg(x(_804)),and(x(_804),y(_804))),
              or(and(x(_804),y(_804)),neg(y(_804)))),neg(cin(_804)))
Applying Rule 1A to in1(g5(_804))
Value of and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))),
               neg(cin(_804))),cin(_804)):
         and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))),
               neg(cin(_804))),cin(_804))
Value of or(and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))),
               or(and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))),
               neg(cin(_804)))),and(or(and(or(neg(x(_804)),
               and(x(_804),y(_804))),or(and(x(_804),y(_804)),
               neg(y(_804)))),neg(cin(_804))),cin(_804))):
         or(and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))),
```

```
               or(and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))),neg(cin(_804)))),
               and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))),neg(cin(_804))),
               cin(_804)))
Does or(and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))),
               or(and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))),neg(cin(_804)))),
               and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))),neg(cin(_804))),
               cin(_804))) =
      xor(xor(x(_804),y(_804)),cin(_804))
or(and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),and(or(neg(x(_804)),
              neg(y(_804))),y(_804))),or(and(or(neg(x(_804)),
              and(x(_804),y(_804))),or(and(x(_804),y(_804)),
              neg(y(_804)))),neg(cin(_804)))),and(or(and(or(neg(x(_804)),
              and(x(_804),y(_804))),or(and(x(_804),y(_804)),neg(y(_804)))),
              neg(cin(_804))),cin(_804))) =
     xor(xor(x(_804),y(_804)),cin(_804))
By Boolean Expansion
output list is [outcarry(_855)]
derived list is[outcarry(_932)]
Outnum is1
Derived  number is1
<<< Success! Behavior of faddxor meets its specification.

yes
| ?- ['qctrld.pro'].
[consulting /usr/users/ela/labovitz/newverify/qctrld.pro...]
 [consulting /usr/users/ela/labovitz/newverify/qops.pro...]
[WARNING, goal failed:  :- unknown(trace,fail)]
 [qops.pro consulted 0.350 sec -4,520 bytes]
 [consulting /usr/users/ela/labovitz/newverify/eval.pro...]
[WARNING: Singleton variables, clause 3 of evaluate_brown/2: F]
 [eval.pro consulted 0.717 sec 0 bytes]
 [consulting /usr/users/ela/labovitz/newverify/derbeh.pro...]
[WARNING: Singleton variables, clause 1 of derive_behaviors/3: Spec_Behavior]
[WARNING: Singleton variables, clause 2 of derive_behaviors/3: Spec_Behavior]
[WARNING: Clauses for derive_behavior/3 are not together in the source file]
[WARNING: Singleton variables, clause 1 of derive_behavior/3: F]
[WARNING: Singleton variables, clause 2 of derive_behavior/3: F]
[WARNING: Singleton variables, clause 8 of derive_behavior/3: Module]
 [derbeh.pro consulted 0.983 sec 0 bytes]
 [consulting /usr/users/ela/labovitz/newverify/derstate.pro...]
```

```
[WARNING: Singleton variables, clause 1 of derive_states/3: Type, Part]
[WARNING: Singleton variables, clause 1 of replace_all/5: Part]
[WARNING: Singleton variables, clause 2 of replace_all/5: Module, Old, New]
[WARNING: Singleton variables, clause 1 of replace/4: Old, New]
[WARNING: Singleton variables, clause 2 of replace/4: Old, New]
[WARNING: Singleton variables, clause 3 of replace/4: New, Arg1]
[WARNING: Singleton variables, clause 11 of replace/4: Old, New]
 [derstate.pro consulted 0.950 sec 0 bytes]
 [consulting /usr/users/ela/labovitz/newverify/counter.pro...]
[WARNING: Singleton variables, clause 1 of port/4: AnInc]
[WARNING: Singleton variables, clause 2 of port/4: AnInc]
[WARNING: Clauses for module_name/1 are not together in the source file]
[WARNING: Clauses for port/4 are not together in the source file]
[WARNING: Singleton variables, clause 1 of port/4: AMux]
[WARNING: Singleton variables, clause 2 of port/4: AMux]
[WARNING: Singleton variables, clause 3 of port/4: AMux]
[WARNING: Singleton variables, clause 4 of port/4: AMux]
[WARNING: Clauses for output_eqn/2 are not together in the source file]
[WARNING: Singleton variables, clause 1 of port/4: AReg]
[WARNING: Singleton variables, clause 2 of port/4: AReg]
[WARNING: Singleton variables, clause 1 of state_of/3: AReg]
[WARNING: Singleton variables, clause 1 of port/4: ACounter]
[WARNING: Singleton variables, clause 2 of port/4: ACounter]
[WARNING: Singleton variables, clause 3 of port/4: ACounter]
[WARNING: Singleton variables, clause 1 of part/3: ACounter]
[WARNING: Singleton variables, clause 2 of part/3: ACounter]
[WARNING: Singleton variables, clause 3 of part/3: ACounter]
[WARNING: Clauses for state_of/3 are not together in the source file]
[WARNING: Singleton variables, clause 1 of state_of/3: ACounter]
[WARNING: Clauses for state_eqn/2 are not together in the source file]
 [counter.pro consulted 1.217 sec 3,704 bytes]
 [consulting /usr/users/ela/labovitz/newverify/boole2.pro...]
[WARNING: Singleton variables, clause 1 of remove_x_1/3: X]
[WARNING: Singleton variables, clause 2 of remove_x_1/3: X]
[WARNING: Singleton variables, clause 3 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 4 of remove_x_1/3: Arg, Arg1, Arg2]
[WARNING: Singleton variables, clause 5 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 6 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 1 of remove_x_0/3: X]
[WARNING: Singleton variables, clause 2 of remove_x_0/3: X]
[WARNING: Singleton variables, clause 3 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 4 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 5 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 6 of remove_x_0/3: Arg2]
 [boole2.pro consulted 1.850 sec 0 bytes]
```

```
 [consulting /usr/users/ela/labovitz/newverify/eqbeh.pro...]
[WARNING: Singleton variables, clause 1 of eqb/3: M]
[WARNING: Singleton variables, clause 2 of eqb/3: M]
[WARNING: Clauses for eqb/3 are not together in the source file]
 [eqbeh.pro consulted 0.400 sec 0 bytes]
 [consulting /usr/users/ela/labovitz/newverify/verify.pro...]
 [verify.pro consulted 0.833 sec 0 bytes]
[qctrld.pro consulted 7.816 sec -616 bytes]

yes
| ?- verify(counter).
>>> Attempting to verify counter>>>
>>>mux primitive (needs no verification)>>>
>>>reg primitive (needs no verification)>>>
>>>inc primitive (needs no verification)>>>
component list is [inc]
Applying Rule 1B to out(_681)
Applying Rule 2A to out(regA(_681))
reg's output equation:
   out(regA(_681)) := contents(regA(_681))
Applying default Rule to contents(regA(_681))
Derived Behavior:  contents(regA(_681))
Rule 1
Rule4
Rule4
Rule4
Value of count(_681):
          count(_681)
Value of count(_681):
          count(_681)
Value of count(_681):
          count(_681)
Substituted Behavior:  count(_681)
output list is [out(_729)]
derived list is[out(_784)]
Outnum is1
Derived  number is1
Applying Rule 1B to in(regA(_1277))
Applying Rule 2A to out(muxA(_1277))
mux's output equation:
   out(muxA(_1277)) := if(switch(muxA(_1277)),in1(muxA(_1277)),
             in0(muxA(_1277)))
Applying Rule 6 to if(switch(muxA(_1277)),in1(muxA(_1277)),i  muxA(_1277)))
Applying Rule 1A to switch(muxA(_1277))
Applying Rule 1A to in1(muxA(_1277))
```

```
Applying Rule 1B to in0(muxA(_1277))
Applying Rule 2A to out(incA(_1277))
inc's output equation:
   out(incA(_1277)) := 1+in(incA(_1277))
Applying Rule 7 to 1+in(incA(_1277))
Applying default Rule to 1
Applying Rule 1B to in(incA(_1277))
Applying Rule 2A to out(regA(_1277))
reg's output equation:
   out(regA(_1277)) := contents(regA(_1277))
Applying default Rule to contents(regA(_1277))
Value of 1+contents(regA(_1277)):
          1+contents(regA(_1277))
Value of if(ctrl(_1277),in(_1277),1+contents(regA(_1277))):
          if(ctrl(_1277),in(_1277),1+contents(regA(_1277)))
Derived Behavior:  if(ctrl(_1277),in(_1277),1+contents(regA(_1277)))
Rule if
Rule4
Rule4
Rule +
Rule2
Rule 1
Rule if
Rule4
Rule4
Rule +
Rule2
Rule4
Rule if
Rule4
Rule3
Rule struct
Rule +
Rule2
Rule4
Rule if
Rule4
Rule4
Rule +
Rule2
Rule4
Value of if(ctrl(_1277),in(_1277),1+count(_1277)):
          if(ctrl(_1277),in(_1277),1+count(_1277))
Value of if(ctrl(_1277),in(_1277),1+count(_1277)):
          if(ctrl(_1277),in(_1277),1+count(_1277))
```

```
Value of if(ctrl(_1277),in(_1277),1+count(_1277)):
          if(ctrl(_1277),in(_1277),1+count(_1277))
Substituted Behavior:  if(ctrl(_1277),in(_1277),1+count(_1277))
Value of if(ctrl(_1277),in(_1277),1+count(_1277)):
          if(ctrl(_1277),in(_1277),1+count(_1277))
Value of if(ctrl(_1277),in(_1277),count(_1277)+1):
          if(ctrl(_1277),in(_1277),1+count(_1277))
Derived behavior is: if(ctrl(_1277),in(_1277),1+count(_1277))
state list is [count(_1311)]
derived list is[count(_1377)]
Statenum is1
Derived  number is1
<<< Success! Behavior of counter meets its specification.

yes
| ?- ^D
csh> exit
csh>
script done on Fri Aug  2 14:32:36 1991
```

B.2 Sample Verification Runs Using New AFIT_VERIFY

B.2.1 Verification of One-Bit Full Adder faddxor.pl

```
Script started on Mon Nov 25 09:11:38 1991
csh> AFIT_Verify
                    Welcome to AFIT_VERIFY!
                    =======================

            (Type ? at any prompt if you require help)



              Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: r
Choices: [xor,faddxor,counter,inv]: f
Should this verification run be executed in TERSE mode? [yes]: y
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/multdyn.pl...]
 [multdyn.pl consulted 0.333 sec 0 bytes]
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/faddxor.pl...]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/primitive.pl...]
  [primitive.pl consulted 0.516 sec 2,932 bytes]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/xor.pl...]
  [xor.pl consulted 0.483 sec 1,556 bytes]
 [faddxor.pl consulted 1.833 sec 7,404 bytes]
Component file faddxor loaded....
--- Beginning verification of module faddxor


>>> Attempting to verify non-primitive module faddxor>>>
>>>nand2 primitive (needs no verification)>>>

>>>nand2 previously verified >>>

>>>nand2 previously verified >>>
```

```
>>> Attempting to verify non-primitive module xor>>>
>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

+> Module xor has verified submodules: [nand2]

Applying Derive_Behavior Rule 2A to out(g4(_8432)) of
   primitive component nand2:
 nand2's output equation:
   out(g4(_8432)) := or(neg(in0(g4(_8432))),neg(in1(g4(_8432))))

Applying Derive_Behavior Rule 2A to out(g2(_8432)) of
   primitive component nand2:
 nand2's output equation:
   out(g2(_8432)) := or(neg(in0(g2(_8432))),neg(in1(g2(_8432))))

Applying Derive_Behavior Rule 2A to out(g1(_8432)) of
   primitive component nand2:
 nand2's output equation:
   out(g1(_8432)) := or(neg(in0(g1(_8432))),neg(in1(g1(_8432))))

Value of neg(or(neg(in0(_8432)),neg(in1(_8432)))):
         and(in0(_8432),in1(_8432))

Value of neg(or(neg(in0(_8432)),and(in0(_8432),in1(_8432)))):
         and(in0(_8432),or(neg(in0(_8432)),neg(in1(_8432))))

Applying Derive_Behavior Rule 2A to out(g3(_8432)) of
   primitive component nand2:
 nand2's output equation:
   out(g3(_8432)) := or(neg(in0(g3(_8432))),neg(in1(g3(_8432))))

Applying Derive_Behavior Rule 2A to out(g1(_8432)) of
   primitive component nand2:
 nand2's output equation:
   out(g1(_8432)) := or(neg(in0(g1(_8432))),neg(in1(g1(_8432))))

Value of neg(or(neg(in0(_8432)),neg(in1(_8432)))):
```

```
         and(in0(_8432),in1(_8432))

Value of neg(or(and(in0(_8432),in1(_8432)),neg(in1(_8432)))):
         and(or(neg(in0(_8432)),neg(in1(_8432))),in1(_8432))


Does or(and(in0(_8432),or(neg(in0(_8432)),neg(in1(_8432)))),
        and(or(neg(in0(_8432)),neg(in1(_8432))),in1(_8432))) =
     or(and(neg(in0(_8432)),in1(_8432)),and(in0(_8432),neg(in1(_8432)))) ???

 or(and(in0(_8432),or(neg(in0(_8432)),neg(in1(_8432)))),
    and(or(neg(in0(_8432)),neg(in1(_8432))),in1(_8432))) =
      or(and(neg(in0(_8432)),in1(_8432)),and(in0(_8432),neg(in1(_8432))))
   By Boolean Expansion


For module xor :
 Specified output list is [out(_8502)]
 Derived output list is   [out(_8541)]
  Number of specified outputs is 1
  Number of derived ouputs is    1

   [out(_8502)] matches with [out(_8541)]

<<< Success! Behavior of xor meets its specification.<<<

>>>xor previously verified >>>

+> Module faddxor has verified submodules: [nand2,xor]

Applying Derive_Behavior Rule 2A to out(g3(_8202)) of
   primitive component nand2:
 nand2's output equation:
   out(g3(_8202)) := or(neg(in0(g3(_8202))),neg(in1(g3(_8202))))

Applying Derive_Behavior Rule 2A to out(g1(_8202)) of
   primitive component nand2:
 nand2's output equation:
   out(g1(_8202)) := or(neg(in0(g1(_8202))),neg(in1(g1(_8202))))

Value of neg(or(neg(x(_8202)),neg(y(_8202)))):
         and(x(_8202),y(_8202))

Applying Derive_Behavior Rule 2A to out(g2(_8202)) of
   primitive component nand2:
```

```
 nand2's output equation:
   out(g2(_8202)) := or(neg(in0(g2(_8202))),neg(in1(g2(_8202))))

Applying Derive_Behavior Rule 2B to out(g4(_8202)) of
   nonprimitive component xor:
 xor's derived behavior:
   out(g4(_8202)) := or(and(in0(g4(_8202)),or(neg(in0(g4(_8202))),
                        neg(in1(g4(_8202))))),
   and(or(neg(in0(g4(_8202))),neg(in1(g4(_8202)))),in1(g4(_8202))))
Value of neg(or(and(x(_8202),or(neg(x(_8202)),neg(y(_8202)))),
             and(or(neg(x(_8202)),neg(y(_8202))),y(_8202)))):
          and(or(neg(x(_8202)),and(x(_8202),y(_8202))),
             or(and(x(_8202),y(_8202)),neg(y(_8202))))

Value of neg(or(neg(cin(_8202)),and(or(neg(x(_8202)),and(x(_8202),y(_8202))),
             or(and(x(_8202),y(_8202)),neg(y(_8202))))))):
          and(cin(_8202),or(and(x(_8202),or(neg(x(_8202)),neg(y(_8202)))),
             and(or(neg(x(_8202)),neg(y(_8202))),y(_8202))))


Does or(and(x(_8202),y(_8202)),and(cin(_8202),or(and(x(_8202),or(neg(x(_8202)),
        neg(y(_8202)))),and(or(neg(x(_8202)),neg(y(_8202))),y(_8202))))) =
     or(and(x(_8202),y(_8202)),and(cin(_8202),xor(x(_8202),y(_8202)))) ???

 or(and(x(_8202),y(_8202)),and(cin(_8202),or(and(x(_8202),or(neg(x(_8202)),
         neg(y(_8202)))),and(or(neg(x(_8202)),neg(y(_8202))),y(_8202))))) =
      or(and(x(_8202),y(_8202)),and(cin(_8202),xor(x(_8202),y(_8202))))
   By Boolean Expansion

Applying Derive_Behavior Rule 2B to out(g5(_8202)) of
   nonprimitive component xor:
 xor's derived behavior:
   out(g5(_8202)) := or(and(in0(g5(_8202)),or(neg(in0(g5(_8202))),
                        neg(in1(g5(_8202))))),
                        and(or(neg(in0(g5(_8202))),
                        neg(in1(g5(_8202)))),in1(g5(_8202))))
Applying Derive_Behavior Rule 2B to out(g4(_8202)) of
   nonprimitive component xor:
 xor's derived behavior:
   out(g4(_8202)) := or(and(in0(g4(_8202)),or(neg(in0(g4(_8202))),
                        neg(in1(g4(_8202))))),and(or(neg(in0(g4(_8202))),
                        neg(in1(g4(_8202)))),in1(g4(_8202))))
Applying Derive_Behavior Rule 2B to out(g4(_8202)) of
   nonprimitive component xor:
 xor's derived behavior:
```

```
   out(g4(_8202)) := or(and(in0(g4(_8202)),or(neg(in0(g4(_8202))),
                        neg(in1(g4(_8202))))),and(or(neg(in0(g4(_8202))),
                        neg(in1(g4(_8202)))),in1(g4(_8202))))
Value of neg(or(and(x(_8202),or(neg(x(_8202)),neg(y(_8202)))),
                and(or(neg(x(_8202)),neg(y(_8202))),y(_8202)))):
         and(or(neg(x(_8202)),and(x(_8202),y(_8202))),
             or(and(x(_8202),y(_8202)),neg(y(_8202))))

Applying Derive_Behavior Rule 2B to out(g4(_8202)) of
   nonprimitive component xor:
 xor's derived behavior:
   out(g4(_8202)) := or(and(in0(g4(_8202)),or(neg(in0(g4(_8202))),
                        neg(in1(g4(_8202))))),and(or(neg(in0(g4(_8202))),
                        neg(in1(g4(_8202)))),in1(g4(_8202))))
Value of neg(or(and(x(_8202),or(neg(x(_8202)),neg(y(_8202)))),
             and(or(neg(x(_8202)),neg(y(_8202))),y(_8202)))):
         and(or(neg(x(_8202)),and(x(_8202),y(_8202))),
             or(and(x(_8202),y(_8202)),neg(y(_8202))))


Does or(and(or(and(x(_8202),or(neg(x(_8202)),neg(y(_8202)))),
        and(or(neg(x(_8202)),neg(y(_8202))),y(_8202))),or(and(or(neg(x(_8202)),
        and(x(_8202),y(_8202))),or(and(x(_8202),y(_8202)),neg(y(_8202)))),
        neg(cin(_8202)))),and(or(and(or(neg(x(_8202)),and(x(_8202),y(_8202))),
        or(and(x(_8202),y(_8202)),neg(y(_8202)))),neg(cin(_8202))),
        cin(_8202))) =
     xor(xor(x(_8202),y(_8202)),cin(_8202)) ???

 or(and(or(and(x(_8202),or(neg(x(_8202)),neg(y(_8202)))),and(or(neg(x(_8202)),
        neg(y(_8202))),y(_8202))),or(and(or(neg(x(_8202)),and(x(_8202),
        y(_8202))),or(and(x(_8202),y(_8202)),neg(y(_8202)))),
        neg(cin(_8202)))),and(or(and(or(neg(x(_8202)),and(x(_8202),y(_8202))),
        or(and(x(_8202),y(_8202)),neg(y(_8202)))),neg(cin(_8202))),
        cin(_8202))) =
      xor(xor(x(_8202),y(_8202)),cin(_8202))
   By Boolean Expansion


For module faddxor :
 Specified output list is [outcarry(_8288),outsum(_8272)]
 Derived output list is   [outcarry(_8329),outsum(_8345)]
  Number of specified outputs is 2
  Number of derived ouputs is    2

   [outcarry(_8288),outsum(_8272)] matches with [outcarry(_8329),outsum(_8345)]
```

```
<<< Success! Behavior of faddxor meets its specification.<<<

>>>> Component faddxor verified! <<<<


            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: l

The part "faddxor" has been previously verified during this session.
The part "xor" has been previously verified during this session.
The part "nand2" has been previously verified during this session.


            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: h
Do you really want to halt Prolog? y/n  [n]? y
csh> exit
csh>
script done on Mon Nov 25 09:12:18 1991
```

B.2.2 Verification of Exclusive Or xor.pl *(Verbose Mode)*

```
Script started on Mon Nov 25 09:12:25 1991
csh> AFIT_Verify
                    Welcome to AFIT_VERIFY!
                    =======================

            (Type ? at any prompt if you require help)



              Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: r
Choices: [xor,faddxor,counter,inv]: ?
Please enter one of these constants:
[xor,faddxor,counter,inv]
followed by a RETURN.  Do not add a full stop.
Choices: [xor,faddxor,counter,inv]: xor
Should this verification run be executed in TERSE mode? [yes]: no
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/multdyn.pl...]
 [multdyn.pl consulted 0.367 sec 0 bytes]
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/xor.pl...]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/primitive.pl...]
  [primitive.pl consulted 0.533 sec 2,932 bytes]
 [xor.pl consulted 1.100 sec 4,908 bytes]
Component file xor loaded....
--- Beginning verification of module xor


>>> Attempting to verify non-primitive module xor>>>
>>>nand2 primitive (needs no verification)>>>

>>>nand2 previously verified >>>
```

```
>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

+> Module xor has verified submodules: [nand2]

Applying Derive_Behavior Rule 1B to out(_8019)
Applying Derive_Behavior Rule 2A to out(g4(_8019)) of
   primitive component nand2:
 nand2's output equation:
   out(g4(_8019)) := or(neg(in0(g4(_8019))),neg(in1(g4(_8019))))

Applying Derive_Behavior Rule 5 to or(neg(in0(g4(_8019))),neg(in1(g4(_8019))))
Applying Derive_Behavior Rule 3 to neg(in0(g4(_8019)))
Applying Derive_Behavior Rule 1B to in0(g4(_8019))
Applying Derive_Behavior Rule 2A to out(g2(_8019)) of
   primitive component nand2:
 nand2's output equation:
   out(g2(_8019)) := or(neg(in0(g2(_8019))),neg(in1(g2(_8019))))

Applying Derive_Behavior Rule 5 to or(neg(in0(g2(_8019))),neg(in1(g2(_8019))))
Applying Derive_Behavior Rule 3 to neg(in0(g2(_8019)))
Applying Derive_Behavior Rule 1A to in0(g2(_8019))

Value of neg(in0(_8019)):
         is already canonical.
Applying Derive_Behavior Rule 3 to neg(in1(g2(_8019)))
Applying Derive_Behavior Rule 1B to in1(g2(_8019))
Applying Derive_Behavior Rule 2A to out(g1(_8019)) of
   primitive component nand2:
 nand2's output equation:
   out(g1(_8019)) := or(neg(in0(g1(_8019))),neg(in1(g1(_8019))))

Applying Derive_Behavior Rule 5 to or(neg(in0(g1(_8019))),neg(in1(g1(_8019))))
Applying Derive_Behavior Rule 3 to neg(in0(g1(_8019)))
Applying Derive_Behavior Rule 1A to in0(g1(_8019))

Value of neg(in0(_8019)):
         is already canonical.
Applying Derive_Behavior Rule 3 to neg(in1(g1(_8019)))
Applying Derive_Behavior Rule 1A to in1(g1(_8019))

Value of neg(in1(_8019)):
         is already canonical.
Value of or(neg(in0(_8019)),neg(in1(_8019))):
```

```
         is already canonical.
Value of neg(or(neg(in0(_8019)),neg(in1(_8019)))):
         and(in0(_8019),in1(_8019))

Value of or(neg(in0(_8019)),and(in0(_8019),in1(_8019))):
         is already canonical.
Value of neg(or(neg(in0(_8019)),and(in0(_8019),in1(_8019)))):
         and(in0(_8019),or(neg(in0(_8019)),neg(in1(_8019))))

Applying Derive_Behavior Rule 3 to neg(in1(g4(_8019)))
Applying Derive_Behavior Rule 1B to in1(g4(_8019))
Applying Derive_Behavior Rule 2A to out(g3(_8019)) of
   primitive component nand2:
 nand2's output equation:
   out(g3(_8019)) := or(neg(in0(g3(_8019))),neg(in1(g3(_8019))))

Applying Derive_Behavior Rule 5 to or(neg(in0(g3(_8019))),neg(in1(g3(_8019))))
Applying Derive_Behavior Rule 3 to neg(in0(g3(_8019)))
Applying Derive_Behavior Rule 1B to in0(g3(_8019))
Applying Derive_Behavior Rule 2A to out(g1(_8019)) of
   primitive component nand2:
 nand2's output equation:
   out(g1(_8019)) := or(neg(in0(g1(_8019))),neg(in1(g1(_8019))))

Applying Derive_Behavior Rule 5 to or(neg(in0(g1(_8019))),neg(in1(g1(_8019))))
Applying Derive_Behavior Rule 3 to neg(in0(g1(_8019)))
Applying Derive_Behavior Rule 1A to in0(g1(_8019))

Value of neg(in0(_8019)):
         is already canonical.
Applying Derive_Behavior Rule 3 to neg(in1(g1(_8019)))
Applying Derive_Behavior Rule 1A to in1(g1(_8019))

Value of neg(in1(_8019)):
         is already canonical.
Value of or(neg(in0(_8019)),neg(in1(_8019))):
         is already canonical.
Value of neg(or(neg(in0(_8019)),neg(in1(_8019)))):
         and(in0(_8019),in1(_8019))

Applying Derive_Behavior Rule 3 to neg(in1(g3(_8019)))
Applying Derive_Behavior Rule 1A to in1(g3(_8019))

Value of neg(in1(_8019)):
         is already canonical.
```

```
Value of or(and(in0(_8019),in1(_8019)),neg(in1(_8019))):
         is already canonical.
Value of neg(or(and(in0(_8019),in1(_8019)),neg(in1(_8019)))):
         and(or(neg(in0(_8019)),neg(in1(_8019))),in1(_8019))

Value of or(and(in0(_8019),or(neg(in0(_8019)),neg(in1(_8019)))),
            and(or(neg(in0(_8019)),neg(in1(_8019))),in1(_8019))):
         is already canonical.

Does or(and(in0(_8019),or(neg(in0(_8019)),neg(in1(_8019)))),
        and(or(neg(in0(_8019)),neg(in1(_8019))),in1(_8019))) =
     or(and(neg(in0(_8019)),in1(_8019)),and(in0(_8019),neg(in1(_8019)))) ???

 or(and(in0(_8019),or(neg(in0(_8019)),neg(in1(_8019)))),
        and(or(neg(in0(_8019)),neg(in1(_8019))),in1(_8019))) =
 or(and(neg(in0(_8019)),in1(_8019)),and(in0(_8019),neg(in1(_8019))))
   By Boolean Expansion


For module xor :
 Specified output list is [out(_8089)]
 Derived output list is   [out(_8128)]
  Number of specified outputs is 1
  Number of derived ouputs is    1

   [out(_8089)] matches with [out(_8128)]

<<< Success! Behavior of xor meets its specification.<<<

>>>> Component xor verified! <<<<



            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)
```

```
Enter your choice: preload, reverify, list, verify, insert, extract, halt: l

The part "xor" has been previously verified during this session.
The part "nand2" has been previously verified during this session.


            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: halt
Do you really want to halt Prolog? y/n  [n]? y
csh> exit
csh>
script done on Mon Nov 25 09:13:03 1991
```

B.2.3 Verification of Exclusive Or xor.pl *(Terse Mode)*

```
Script started on Mon Nov 25 09:13:59 1991
csh> AFIT_Verify
                  Welcome to AFIT_VERIFY!
                  =======================

           (Type ? at any prompt if you require help)



             Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: r
Choices: [xor,faddxor,counter,inv]: xor
Should this verification run be executed in TERSE mode? [yes]:
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/multdyn.pl...]
 [multdyn.pl consulted 0.350 sec 0 bytes]
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/xor.pl...]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/primitive.pl...]
  [primitive.pl consulted 0.517 sec 2,932 bytes]
 [xor.pl consulted 1.066 sec 4,908 bytes]
Component file xor loaded....
--- Beginning verification of module xor


>>> Attempting to verify non-primitive module xor>>>
>>>nand2 primitive (needs no verification)>>>

>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

>>>nand2 previously verified >>>
```

```
        and(or(neg(in0(_8021)),neg(in1(_8021))),in1(_8021))) =
      or(and(neg(in0(_8021)),in1(_8021)),and(in0(_8021),neg(in1(_8021))))
   By Boolean Expansion


For module xor :
 Specified output list is [out(_8091)]
 Derived output list is   [out(_8130)]
  Number of specified outputs is 1
  Number of derived ouputs is    1

   [out(_8091)] matches with [out(_8130)]

<<< Success! Behavior of xor meets its specification.<<<

>>>> Component xor verified! <<<<



            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: halt
Do you really want to halt Prolog? y/n  [n]? y
csh> exit
csh>
script done on Mon Nov 25 09:14:23 1991
```

B.2.4 Verification of The Three-, Four-, and Five-Input NAND Implementations nand3.pl, nand4.pl, *and* nand5.pl

```
Script started on Mon Nov 25 09:15:11 1991
csh> AFIT_Verify
                    Welcome to AFIT_VERIFY!
                    =======================

            (Type ? at any prompt if you require help)



              Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: v
Name of module (file) to be verified (do not include .pl suffix): nand5
Should this verification run be executed in TERSE mode? [yes]:
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/multdyn.pl...]
 [multdyn.pl consulted 0.366 sec 0 bytes]
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Components/nand5.pl...]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Components/nand3.pl...]
   [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/primitive.pl...]
   [primitive.pl consulted 0.517 sec 2,900 bytes]
   [consulting /usr/users/ela/labovitz/NewVerify/Work/Components/inv.pl...]
   [inv.pl consulted 0.267 sec 800 bytes]
  [nand3.pl consulted 1.383 sec 5,696 bytes]
 [nand5.pl consulted 1.917 sec 7,808 bytes]
Component file nand5 loaded....
--- Beginning verification of module nand5


>>> Attempting to verify non-primitive module nand5>>>

>>> Attempting to verify non-primitive module nand3>>>
>>>nand2 primitive (needs no verification)>>>
```

```
+> Module xor has verified submodules: [nand2]

Applying Derive_Behavior Rule 2A to out(g4(_8021)) of
   primitive component nand2:
 nand2's output equation:
   out(g4(_8021)) := or(neg(in0(g4(_8021))),neg(in1(g4(_8021))))

Applying Derive_Behavior Rule 2A to out(g2(_8021)) of
   primitive component nand2:
 nand2's output equation:
   out(g2(_8021)) := or(neg(in0(g2(_8021))),neg(in1(g2(_8021))))

Applying Derive_Behavior Rule 2A to out(g1(_8021)) of
   primitive component nand2:
 nand2's output equation:
   out(g1(_8021)) := or(neg(in0(g1(_8021))),neg(in1(g1(_8021))))

Value of neg(or(neg(in0(_8021)),neg(in1(_8021)))):
         and(in0(_8021),in1(_8021))

Value of neg(or(neg(in0(_8021)),and(in0(_8021),in1(_8021)))):
         and(in0(_8021),or(neg(in0(_8021)),neg(in1(_8021))))

Applying Derive_Behavior Rule 2A to out(g3(_8021)) of
   primitive component nand2:
 nand2's output equation:
   out(g3(_8021)) := or(neg(in0(g3(_8021))),neg(in1(g3(_8021))))

Applying Derive_Behavior Rule 2A to out(g1(_8021)) of
   primitive component nand2:
 nand2's output equation:
   out(g1(_8021)) := or(neg(in0(g1(_8021))),neg(in1(g1(_8021))))

Value of neg(or(neg(in0(_8021)),neg(in1(_8021)))):
         and(in0(_8021),in1(_8021))

Value of neg(or(and(in0(_8021),in1(_8021)),neg(in1(_8021)))):
         and(or(neg(in0(_8021)),neg(in1(_8021))),in1(_8021))


Does or(and(in0(_8021),or(neg(in0(_8021)),neg(in1(_8021)))),
        and(or(neg(in0(_8021)),neg(in1(_8021))),in1(_8021))) =
     or(and(neg(in0(_8021)),in1(_8021)),and(in0(_8021),neg(in1(_8021)))) ???

 or(and(in0(_8021),or(neg(in0(_8021)),neg(in1(_8021)))),
```

```
>>>nand2 previously verified >>>


>>> Attempting to verify non-primitive module inv>>>
>>>nand2 previously verified >>>

+> Module inv has verified submodules: [nand2]

Applying Derive_Behavior Rule 2A to out(g1(_7582)) of
   primitive component nand2:
 nand2's output equation:
   out(g1(_7582)) := or(neg(in0(g1(_7582))),neg(in1(g1(_7582))))


Does neg(in(_7582)) =
     neg(in(_7582)) ???

 or(neg(in(_7582)),neg(in(_7582))) =
      neg(in(_7582))
   By Boolean Expansion


For module inv :
 Specified output list is [out(_7652)]
 Derived output list is   [out(_7691)]
  Number of specified outputs is 1
  Number of derived ouputs is    1

   [out(_7652)] matches with [out(_7691)]

<<< Success! Behavior of inv meets its specification.<<<

+> Module nand3 has verified submodules: [inv,nand2]

Applying Derive_Behavior Rule 2A to out(nand2_2(_7370)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_2(_7370)) := or(neg(in0(nand2_2(_7370))),
                             neg(in1(nand2_2(_7370))))

Applying Derive_Behavior Rule 2B to out(inv0(_7370)) of
   nonprimitive component inv:
 inv's derived behavior:
   out(inv0(_7370)) := or(neg(in(inv0(_7370))),neg(in(inv0(_7370))))
```

```
Applying Derive_Behavior Rule 2A to out(nand2_1(_7370)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_1(_7370)) := or(neg(in0(nand2_1(_7370))),
                             neg(in1(nand2_1(_7370))))

Value of neg(or(neg(in0(_7370)),neg(in1(_7370)))):
         and(in0(_7370),in1(_7370))

Applying Derive_Behavior Rule 2A to out(nand2_1(_7370)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_1(_7370)) := or(neg(in0(nand2_1(_7370))),
                             neg(in1(nand2_1(_7370))))

Value of neg(or(neg(in0(_7370)),neg(in1(_7370)))):
         and(in0(_7370),in1(_7370))

Value of neg(or(and(in0(_7370),in1(_7370)),and(in0(_7370),in1(_7370)))):
         and(or(neg(in0(_7370)),neg(in1(_7370))),or(neg(in0(_7370)),
             neg(in1(_7370))))


Does or(or(neg(in0(_7370)),neg(in1(_7370))),neg(in2(_7370))) =
     neg(and(and(in0(_7370),in1(_7370)),in2(_7370))) ???

 or(and(or(neg(in0(_7370)),neg(in1(_7370))),or(neg(in0(_7370)),
           neg(in1(_7370)))),neg(in2(_7370))) =
      neg(and(and(in0(_7370),in1(_7370)),in2(_7370)))
   By Boolean Expansion


For module nand3 :
 Specified output list is [out(_7440)]
 Derived output list is   [out(_7479)]
  Number of specified outputs is 1
  Number of derived ouputs is    1

   [out(_7440)] matches with [out(_7479)]

<<< Success! Behavior of nand3 meets its specification.<<<

>>>nand3 previously verified >>>

>>>inv previously verified >>>
```

```
+> Module nand5 has verified submodules: [inv,nand3]

Applying Derive_Behavior Rule 2B to out(nand3_2(_7062)) of
   nonprimitive component nand3:
 nand3's derived behavior:
   out(nand3_2(_7062)) := or(and(or(neg(in0(nand3_2(_7062))),
                             neg(in1(nand3_2(_7062)))),
                             or(neg(in0(nand3_2(_7062))),
                             neg(in1(nand3_2(_7062))))),
                             neg(in2(nand3_2(_7062))))
Applying Derive_Behavior Rule 2B to out(inv0(_7062)) of
   nonprimitive component inv:
 inv's derived behavior:
   out(inv0(_7062)) := or(neg(in(inv0(_7062))),neg(in(inv0(_7062))))
Applying Derive_Behavior Rule 2B to out(nand3_1(_7062)) of
   nonprimitive component nand3:
 nand3's derived behavior:
   out(nand3_1(_7062)) :=   or(and(or(neg(in0(nand3_1(_7062))),
                               neg(in1(nand3_1(_7062)))),
                               or(neg(in0(nand3_1(_7062))),
                               neg(in1(nand3_1(_7062))))),
                               neg(in2(nand3_1(_7062))))
Value of neg(or(and(or(neg(in0(_7062)),neg(in1(_7062))),or(neg(in0(_7062)),
            neg(in1(_7062)))),neg(in2(_7062)))):
         and(or(and(in0(_7062),in1(_7062)),and(in0(_7062),in1(_7062))),in2(_7062))

Applying Derive_Behavior Rule 2B to out(nand3_1(_7062)) of
   nonprimitive component nand3:
 nand3's derived behavior:
   out(nand3_1(_7062)) := or(and(or(neg(in0(nand3_1(_7062))),
                             neg(in1(nand3_1(_7062)))),
                             or(neg(in0(nand3_1(_7062))),
                             neg(in1(nand3_1(_7062))))),
                             neg(in2(nand3_1(_7062))))
Value of neg(or(and(or(neg(in0(_7062)),neg(in1(_7062))),or(neg(in0(_7062)),
            neg(in1(_7062)))),neg(in2(_7062)))):
         and(or(and(in0(_7062),in1(_7062)),and(in0(_7062),
            in1(_7062))),in2(_7062))

Value of neg(or(and(or(and(in0(_7062),in1(_7062)),and(in0(_7062),in1(_7062))),
            in2(_7062)),and(or(and(in0(_7062),in1(_7062)),and(in0(_7062),
            in1(_7062))),in2(_7062)))):
         and(or(and(or(neg(in0(_7062)),neg(in1(_7062))),or(neg(in0(_7062)),
            neg(in1(_7062)))),neg(in2(_7062))),or(and(or(neg(in0(_7062)),
```

```
            neg(in1(_7062))),or(neg(in0(_7062)),neg(in1(_7062)))),
            neg(in2(_7062))))

Applying Derive_Behavior Rule 2B to out(inv0(_7062)) of
   nonprimitive component inv:
 inv's derived behavior:
   out(inv0(_7062)) := or(neg(in(inv0(_7062))),neg(in(inv0(_7062))))
Applying Derive_Behavior Rule 2B to out(nand3_1(_7062)) of
   nonprimitive component nand3:
 nand3's derived behavior:
   out(nand3_1(_7062)) := or(and(or(neg(in0(nand3_1(_7062))),
                             neg(in1(nand3_1(_7062)))),
                             or(neg(in0(nand3_1(_7062))),
                             neg(in1(nand3_1(_7062))))),
                             neg(in2(nand3_1(_7062))))
Value of neg(or(and(or(neg(in0(_7062)),neg(in1(_7062))),or(neg(in0(_7062)),
            neg(in1(_7062)))),neg(in2(_7062)))):
         and(or(and(in0(_7062),in1(_7062)),and(in0(_7062),in1(_7062))),in2(_7062))

Applying Derive_Behavior Rule 2B to out(nand3_1(_7062)) of
   nonprimitive component nand3:
 nand3's derived behavior:
   out(nand3_1(_7062)) := or(and(or(neg(in0(nand3_1(_7062))),
                             neg(in1(nand3_1(_7062)))),
                             or(neg(in0(nand3_1(_7062))),
                             neg(in1(nand3_1(_7062))))),
                             neg(in2(nand3_1(_7062))))
Value of neg(or(and(or(neg(in0(_7062)),neg(in1(_7062))),
            or(neg(in0(_7062)),neg(in1(_7062)))),neg(in2(_7062)))):
         and(or(and(in0(_7062),in1(_7062)),and(in0(_7062),in1(_7062))),in2(_7062))

Value of neg(or(and(or(and(in0(_7062),in1(_7062)),and(in0(_7062),
            in1(_7062))),in2(_7062)),and(or(and(in0(_7062),in1(_7062)),
            and(in0(_7062),in1(_7062))),in2(_7062)))):
         and(or(and(or(neg(in0(_7062)),neg(in1(_7062))),or(neg(in0(_7062)),
            neg(in1(_7062)))),neg(in2(_7062))),or(and(or(neg(in0(_7062)),
            neg(in1(_7062))),or(neg(in0(_7062)),neg(in1(_7062)))),
            neg(in2(_7062))))


Does or(or(or(or(neg(in0(_7062)),neg(in1(_7062))),neg(in2(_7062))),
        neg(in3(_7062))),neg(in4(_7062))) =
     neg(and(and(in0(_7062),in1(_7062)),and(in2(_7062),
        and(in3(_7062),in4(_7062))))) ???
```

```
 or(and(or(and(or(and(or(neg(in0(_7062)),neg(in1(_7062))),
        or(neg(in0(_7062)),neg(in1(_7062)))),neg(in2(_7062))),
        or(and(or(neg(in0(_7062)),neg(in1(_7062))),
        or(neg(in0(_7062)),neg(in1(_7062)))),neg(in2(_7062)))),
        neg(in3(_7062))),or(and(or(and(or(neg(in0(_7062)),
        neg(in1(_7062))),or(neg(in0(_7062)),neg(in1(_7062)))),
        neg(in2(_7062))),or(and(or(neg(in0(_7062)),neg(in1(_7062))),
        or(neg(in0(_7062)),neg(in1(_7062)))),neg(in2(_7062)))),
        neg(in3(_7062)))),neg(in4(_7062))) =
 neg(and(and(in0(_7062),in1(_7062)),and(in2(_7062),
        and(in3(_7062),in4(_7062)))))
   By Boolean Expansion


For module nand5 :
 Specified output list is [out(_7132)]
 Derived output list is   [out(_7171)]
  Number of specified outputs is 1
  Number of derived ouputs is    1

   [out(_7132)] matches with [out(_7171)]

<<< Success! Behavior of nand5 meets its specification.<<<

>>>> Component nand5 verified! <<<<

Should this component be inserted into the library?  [no]:


            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: v
Name of module (file) to be verified (do not include .pl suffix): nand3
Should this verification run be executed in TERSE mode? [yes]:
```

```
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/multdyn.pl...]
 [multdyn.pl consulted 0.467 sec -6,264 bytes]
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Components/nand3.pl...]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/primitive.pl...]
  [primitive.pl consulted 0.533 sec 2,632 bytes]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Components/inv.pl...]
  [inv.pl consulted 0.284 sec 640 bytes]
 [nand3.pl consulted 1.467 sec 4,744 bytes]
Component file nand3 loaded....
--- Beginning verification of module nand3


>>>> Component nand3 already verified! <<<<



            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: v
Name of module (file) to be verified (do not include .pl suffix): nand4
Should this verification run be executed in TERSE mode? [yes]:
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/multdyn.pl...]
 [multdyn.pl consulted 0.433 sec -4,004 bytes]
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Components/nand4.pl...]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/primitive.pl...]
  [primitive.pl consulted 0.533 sec 2,632 bytes]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Components/inv.pl...]
  [inv.pl consulted 0.300 sec 640 bytes]
 [nand4.pl consulted 1.550 sec 5,380 bytes]
Component file nand4 loaded....
--- Beginning verification of module nand4


>>> Attempting to verify non-primitive module nand4>>>
```

```
>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

>>>inv previously verified >>>

>>>inv previously verified >>>

+> Module nand4 has verified submodules: [inv,nand2]

Applying Derive_Behavior Rule 2A to out(nand2_2(_20111)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_2(_20111)) := or(neg(in0(nand2_2(_20111))),
                              neg(in1(nand2_2(_20111))))

Applying Derive_Behavior Rule 2B to out(inv0(_20111)) of
   nonprimitive component inv:
 inv's derived behavior:
   out(inv0(_20111)) := or(neg(in(inv0(_20111))),neg(in(inv0(_20111))))
Applying Derive_Behavior Rule 2A to out(nand2_1(_20111)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_1(_20111)) := or(neg(in0(nand2_1(_20111))),
                              neg(in1(nand2_1(_20111))))

Value of neg(or(neg(in0(_20111)),neg(in1(_20111)))):
         and(in0(_20111),in1(_20111))

Applying Derive_Behavior Rule 2A to out(nand2_1(_20111)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_1(_20111)) := or(neg(in0(nand2_1(_20111))),
                              neg(in1(nand2_1(_20111))))

Value of neg(or(neg(in0(_20111)),neg(in1(_20111)))):
         and(in0(_20111),in1(_20111))

Value of neg(or(and(in0(_20111),in1(_20111)),and(in0(_20111),in1(_20111)))):
         and(or(neg(in0(_20111)),neg(in1(_20111))),or(neg(in0(_20111)),
             neg(in1(_20111))))

Applying Derive_Behavior Rule 2B to out(inv1(_20111)) of
```

```
   nonprimitive component inv:
 inv's derived behavior:
   out(inv1(_20111)) := or(neg(in(inv1(_20111))),neg(in(inv1(_20111))))
Applying Derive_Behavior Rule 2A to out(nand2_3(_20111)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_3(_20111)) := or(neg(in0(nand2_3(_20111))),
                              neg(in1(nand2_3(_20111))))

Value of neg(or(neg(in2(_20111)),neg(in3(_20111)))):
         and(in2(_20111),in3(_20111))

Applying Derive_Behavior Rule 2A to out(nand2_3(_20111)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_3(_20111)) := or(neg(in0(nand2_3(_20111))),
                              neg(in1(nand2_3(_20111))))

Value of neg(or(neg(in2(_20111)),neg(in3(_20111)))):
         and(in2(_20111),in3(_20111))

Value of neg(or(and(in2(_20111),in3(_20111)),and(in2(_20111),in3(_20111)))):
         and(or(neg(in2(_20111)),neg(in3(_20111))),
             or(neg(in2(_20111)),neg(in3(_20111))))


Does or(or(neg(in0(_20111)),neg(in1(_20111))),or(neg(in2(_20111)),
        neg(in3(_20111)))) =
     neg(and(and(in0(_20111),in1(_20111)),and(in2(_20111),in3(_20111)))) ???

 or(and(or(neg(in0(_20111)),neg(in1(_20111))),or(neg(in0(_20111)),
          neg(in1(_20111)))),and(or(neg(in2(_20111)),neg(in3(_20111))),
          or(neg(in2(_20111)),neg(in3(_20111))))) =
      neg(and(and(in0(_20111),in1(_20111)),and(in2(_20111),in3(_20111))))
   By Boolean Expansion


For module nand4 :
 Specified output list is [out(_20181)]
 Derived output list is   [out(_20220)]
  Number of specified outputs is 1
  Number of derived ouputs is    1

   [out(_20181)] matches with [out(_20220)]
```

```
<<< Success! Behavior of nand4 meets its specification.<<<

>>>> Component nand4 verified! <<<<

Should this component be inserted into the library?  [no]:


            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: l

The part "nand4" has been previously verified during this session.
The part "nand5" has been previously verified during this session.
The part "nand3" has been previously verified during this session.
The part "inv" has been previously verified during this session.
The part "nand2" has been previously verified during this session.


            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: h
Do you really want to halt Prolog? y/n  [n]? y
csh> exit
```

```
csh>
script done on Mon Nov 25 09:17:17 1991
```

B.2.5 Verification of Half Adder halfadd.pl

```
Script started on Mon Nov 25 09:18:50 1991
csh> AFIT_Verify
                    Welcome to AFIT_VERIFY!
                    =======================

            (Type ? at any prompt if you require help)



              Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
         (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
         (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: v
Name of module (file) to be verified (do not include .pl suffix): halfadd
Should this verification run be executed in TERSE mode? [yes]:
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/multdyn.pl...]
 [multdyn.pl consulted 0.367 sec 0 bytes]
 [consulting /usr/users/ela/labovitz/NewVerify/Work/Components/halfadd.pl...]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Parts/primitive.pl...]
  [primitive.pl consulted 0.533 sec 2,900 bytes]
  [consulting /usr/users/ela/labovitz/NewVerify/Work/Components/inv.pl...]
  [inv.pl consulted 0.283 sec 768 bytes]
 [halfadd.pl consulted 1.684 sec 6,732 bytes]
Component file halfadd loaded....
--- Beginning verification of module halfadd


>>> Attempting to verify non-primitive module halfadd>>>

>>> Attempting to verify non-primitive module inv>>>
>>>nand2 primitive (needs no verification)>>>

+> Module inv has verified submodules: [nand2]
```

```
Applying Derive_Behavior Rule 2A to out(g1(_7330)) of
   primitive component nand2:
 nand2's output equation:
   out(g1(_7330)) := or(neg(in0(g1(_7330))),neg(in1(g1(_7330))))


Does neg(in(_7330)) =
     neg(in(_7330)) ???

 or(neg(in(_7330)),neg(in(_7330))) =
      neg(in(_7330))
   By Boolean Expansion


For module inv :
 Specified output list is [out(_7400)]
 Derived output list is   [out(_7439)]
  Number of specified outputs is 1
  Number of derived ouputs is    1

   [out(_7400)] matches with [out(_7439)]

<<< Success! Behavior of inv meets its specification.<<<

>>>inv previously verified >>>

>>>inv previously verified >>>

>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

>>>nand2 previously verified >>>

+> Module halfadd has verified submodules: [inv,nand2]

Applying Derive_Behavior Rule 2A to out(nand2_2(_7178)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_2(_7178)) := or(neg(in0(nand2_2(_7178))),
                             neg(in1(nand2_2(_7178))))

Applying Derive_Behavior Rule 2A to out(nand2_0(_7178)) of
```

```
   primitive component nand2:
 nand2's output equation:
   out(nand2_0(_7178)) := or(neg(in0(nand2_0(_7178))),
                             neg(in1(nand2_0(_7178))))

Applying Derive_Behavior Rule 2B to out(inv_1(_7178)) of
   nonprimitive component inv:
 inv's derived behavior:
   out(inv_1(_7178)) := or(neg(in(inv_1(_7178))),neg(in(inv_1(_7178))))
Value of neg(or(neg(in1(_7178)),neg(in1(_7178)))):
         and(in1(_7178),in1(_7178))

Value of neg(or(neg(in0(_7178)),and(in1(_7178),in1(_7178)))):
         and(in0(_7178),or(neg(in1(_7178)),neg(in1(_7178))))

Applying Derive_Behavior Rule 2A to out(nand2_1(_7178)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_1(_7178)) := or(neg(in0(nand2_1(_7178))),
                             neg(in1(nand2_1(_7178))))

Applying Derive_Behavior Rule 2B to out(inv_0(_7178)) of
   nonprimitive component inv:
 inv's derived behavior:
   out(inv_0(_7178)) := or(neg(in(inv_0(_7178))),neg(in(inv_0(_7178))))
Value of neg(or(neg(in0(_7178)),neg(in0(_7178)))):
         and(in0(_7178),in0(_7178))

Value of neg(or(neg(in1(_7178)),and(in0(_7178),in0(_7178)))):
         and(in1(_7178),or(neg(in0(_7178)),neg(in0(_7178))))


Does or(and(in0(_7178),neg(in1(_7178))),and(in1(_7178),neg(in0(_7178)))) =
     or(and(neg(in0(_7178)),in1(_7178)),and(in0(_7178),neg(in1(_7178)))) ???

 or(and(in0(_7178),or(neg(in1(_7178)),neg(in1(_7178)))),
        and(in1(_7178),or(neg(in0(_7178)),neg(in0(_7178))))) =
      or(and(neg(in0(_7178)),in1(_7178)),and(in0(_7178),neg(in1(_7178))))
   By Boolean Expansion

Applying Derive_Behavior Rule 2B to out(inv_2(_7178)) of
   nonprimitive component inv:
 inv's derived behavior:
   out(inv_2(_7178)) := or(neg(in(inv_2(_7178))),neg(in(inv_2(_7178))))
Applying Derive_Behavior Rule 2A to out(nand2_3(_7178)) of
```

```
   primitive component nand2:
 nand2's output equation:
   out(nand2_3(_7178)) := or(neg(in0(nand2_3(_7178))),
                             neg(in1(nand2_3(_7178))))

Value of neg(or(neg(in0(_7178)),neg(in1(_7178)))):
         and(in0(_7178),in1(_7178))

Applying Derive_Behavior Rule 2A to out(nand2_3(_7178)) of
   primitive component nand2:
 nand2's output equation:
   out(nand2_3(_7178)) := or(neg(in0(nand2_3(_7178))),
                             neg(in1(nand2_3(_7178))))

Value of neg(or(neg(in0(_7178)),neg(in1(_7178)))):
         and(in0(_7178),in1(_7178))


Does and(in0(_7178),in1(_7178)) =
     and(in0(_7178),in1(_7178)) ???

 or(and(in0(_7178),in1(_7178)),and(in0(_7178),in1(_7178))) =
      and(in0(_7178),in1(_7178))
   By Boolean Expansion


For module halfadd :
 Specified output list is [carry(_7248),sum(_7264)]
 Derived output list is   [carry(_7321),sum(_7305)]
  Number of specified outputs is 2
  Number of derived ouputs is    2

   [carry(_7248),sum(_7264)] matches with [carry(_7321),sum(_7305)]

<<< Success! Behavior of halfadd meets its specification.<<<

>>>> Component halfadd verified! <<<<

Should this component be inserted into the library?  [no]: yes


            Performing AFIT_VERIFY Verification!

Select your action from the following choices:
   Preload the previously verified components into the database
```

```
      (This may increase execution speed of a verification run)
   Reverify a component from the component library
   List the nonprimitive components which have been verified this session
   Insert a component into the component library area
   Extract a component from the library area into current directory
   Verify a new component from the current directory
   Halt the program and exit Prolog
      (Note: this option _is_ revocable at the next menu!)

Enter your choice: preload, reverify, list, verify, insert, extract, halt: halt
Do you really want to halt Prolog? y/n  [n]? exit
Please answer Yes or No followed by RETURN
Do you really want to halt Prolog? y/n  [n]? y
csh> exit
csh>
script done on Mon Nov 25 09:21:57 1991
```

Bibliography

1. Barrow, Harry G. "Proving the Correctness of Digital Hardware Designs." *Proceedings of The National Conference on Artificial Intelligence*. 17–21. 1983.
2. Barrow, Harry G. "Proving the Correctness of Digital Hardware Designs," *VLSI Design*, 64–77 (July 1984).
3. Barrow, Harry G. "VERIFY: A Program for Proving Correctness of Digital Hardware Designs," *Artificial Intelligence*, *24*:437–491 (December 1984).
4. Bratko, Ivan. *Prolog Programming for Artificial Intelligence* (Second Edition). Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1990.
5. Brezocnik, Z., B. Horvat and M. Gerkes. "Tool for System Design Verification." *Proceedings of the CompEuro 88 - System Design: Concepts, Methods and Tools*. 100–107. Washington D.C.: IEEE Computer Society Press, 1988.
6. Cohen, Daniel I. A. *Introduction to Computer Theory* (Second Edition). New York: John Wiley & Sons, Inc., 1991.
7. Cohen, Jacques. "View of the Origins and Development of Prolog," *Communications of the ACM*, *31*(1):26–36 (January 1988).
8. Cohn, Avra. "Correctness Properties of the Viper Block Model: The Second Level." *Current Trends in Hardware Verification and Automated Theorem Proving.* edited by Graham Birtwistle and P. A. Subrahmanyam, 1–91, New York: Springer-Verlag, 1989.
9. Dill, David L. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. PhD dissertation, Carnegie Mellon University, February 1988.
10. Dougherty, Edward R. and Charles R. Giardina. *Mathematical Methods for Artificial Intelligence and Autonomous Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1988.
11. Dukes, CPT Michael A. "Formal Verification Using VHDL." PhD Prospectus, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1990.
12. Dukes, Michael Alan and Frank Markham Brown. *Proving Boolean Equivalence with Prolog*. WRDC Technical Report WRDC-TR-90-5006, Wright-Patterson AFB OH: Wright Research and Development Center, February 1990.
13. Eicher, Capt Joseph W. *Logic Programming in Digital Circuit Design*. MS thesis, AFIT/GCE/ENG/91D-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
14. Goguen, Joseph A. "OBJ as a Theorem Prover with Applications to Hardware Verification." *Current Trends in Hardware Verification and Automated Theorem Proving.* edited by Graham Birtwistle and P. A. Subrahmanyam, 218–267, New York: Springer-Verlag, 1989.

15. Grabowiecki, Tadeusz, Adam Pawlak and Wojciech Sakowski. "A University Framework for Correct by Construction IC Design," *Microprocessing and Microprogramming*, *23*:37–43 (1988).

16. Hill, Fredrick J. and Gerald R. Peterson. *Digital Systems: Hardware Organization and Design* (2nd Edition). New York: John Wiley & Sons, Inc., 1978.

17. Hill, Fredrick J. and Gerald R. Peterson. *Introduction to Switching Theory & Logical Design* (3rd Edition). New York: John Wiley & Sons, Inc., 1981.

18. Hillman, Abraham P. and Gerald L. Alexanderson. *A First Undergraduate Course in Abstract Algebra*. Belmont, California: Wadsworth Publishing Company, Inc., 1983.

19. Hopcroft, John E. and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1979.

20. Howell, Gene Edward. *Analysis, Design, and Testing of Gallium Arsenide Digital Radio Frequency Memory Modules*. MS thesis, AFIT/GCE/ENG/91D-05, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

21. Johnsonbaugh, Richard. *Discrete Mathematics*. New York: Macmillan Publishing Company, 1984.

22. Knaus, Rodger. "A Pocket Guide To PROLOG," *AI Expert*, *5*(7):63–64 (January 1991).

23. Kowalski, Robert A. "Early Years of Logic Programming," *Communications of the ACM*, *31*(1):38–43 (January 1988).

24. Liaw, Heh-Tyan, Kim-Thu Tran and Chen-Shang Lin. "VVDS: A Verification/ Diagnosis System For VHDL." *26th ACM/IEEE Design Automation Conference*. 435–440. New York: ACM Press, 1989.

25. Quintus Computer Systems, Inc., Mountain View, CA. *Quintus Prolog Library Manual*, Nov 1987. Version 1.

26. Quintus Computer Systems, Inc., Mountain View, CA. *Quintus Prolog Reference Manual*, Feb 1987. Version 10.

27. Quintus Computer Systems, Inc., Mountain View, CA. *Quintus Prolog System Dependent Features Manual For UNIX Systems*, Mar 1987. Version 10.

28. Quintus Computer Systems, Inc., Mountain View, CA. *Quintus Prolog User's Guide*, Nov 1987. Version 11.

29. Robinson, A. J. "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the Association of Computing Machinery*, *12*:23–41 (January 1965).

30. Sparks, Capt Kevin L. *A Prolog-Based System for Hardware Verification*. MS thesis, AFIT/GCE/ENG/91M-05, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1991.

31. Sterling, Leon and Ehud Shapiro. *The Art Of Prolog: Advanced Programming Techniques*. Cambridge, MA: The MIT Press, 1986.

32. Zycad Corporation. *Zycad System VHDL Reference Manual*, 1990. Revision 2.0a.

Vita

Captain Stuart L. Labovitz was born on April 4, 1963 in New Haven, Connecticut. He attended Manalapan High School, Manalapan, New Jersey, and graduated in June 1981. He received a Bachelor of Science in Electrical Engineering from Lehigh University in June 1985. He was a Distinguished Military Graduate of the Air Force Reserve Officer Training Corps receiving a regular commission as a Second Lieutenant in the USAF. After graduation, he worked as a civilian employee for the US Army Avionics Research and Development Activity, Fort Monmouth, New Jersey, from June 1985 through September 1985. Following entry into active duty in October 1985, he was assigned to the Avionics Laboratory, Air Force Wright Aeronautical Laboratories. During this assignment, he served as Microwave Project Engineer, and then as Project Engineer and Executive Officer for the newly formed Electronic Technology Laboratory, Wright Research and Development Center. He began full-time work on a Masters of Science in Computer Engineering in June 1990, and graduated from this program in December 1991.

Permanent address: 18 Sheffield Drive
Manalapan, NJ 07726-3619